UC Berkeley
UC Berkeley Electronic Theses and Dissertations

Title

Compartmentalizing State Machine Replication

Permalink

https://escholarship.org/uc/item/7h89264s

Author

Whittaker, Michael

Publication Date

2021

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

Compartmentalizing State Machine Replication

by

Michael Whittaker

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Joseph Hellerstein, Chair
Assistant Professor Natacha Crooks
Assistant Professor Barna Saha
Professor Ion Stoica

Summer 2021

Compartmentalizing State Machine Replication

Copyright 2021
by
Michael Whittaker

Abstract

Compartmentalizing State Machine Replication

by

Michael Whittaker

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Joseph Hellerstein, Chair

State machine replication is at the heart of almost every strongly consistent distributed system. In this thesis, we introduce a novel technique called compartmentalization that involves decoupling a protocol into its simplest components and then scaling each component independently. Compartmentalization is simple yet effective. It can be used to increase the throughput of a protocol, to simplify an exiting protocol, or to design new functionality for a protocol. In this thesis specifically, we apply compartmentalization to state machine replication protocols in a number of different ways. We compartmentalize MultiPaxos and increase its throughput by over an order of magnitude. We then compartmentalize a family of complex state machine protocols called generalized multi-leader protocols. The compartmentalization simplifies the protocols and brings clarity to a family of protocols that were previously extremely difficult to understand. Finally, we use compartmentalization to design a new reconfiguration protocol based on Vertical Paxos that state machine replication protocols can use to replace failed machines with new machines without any downtime.

To my friends and family

Contents

Contents ii

List of Figures iv

List of Tables ix

1 Introduction 1

2 Background 4
- 2.1 System Model . . . 4
- 2.2 Paxos . . . 4
- 2.3 MultiPaxos . . . 8
- 2.4 Flexible Paxos . . . 9

3 Compartmentalization 10
- 3.1 MultiPaxos Does Not Scale? . . . 11
- 3.2 Compartmentalizing MultiPaxos . . . 12
- 3.3 Batching . . . 26
- 3.4 Mencius . . . 30
- 3.5 S-Paxos . . . 33
- 3.6 Evaluation . . . 35
- 3.7 Related Work . . . 44

4 Quoracle 48
- 4.1 Definitions . . . 49
- 4.2 Practical Refinements in Quoracle . . . 51
- 4.3 Case Study . . . 58
- 4.4 Lessons Learned . . . 60

5 Bipartisan Paxos 62
- 5.1 Conflict Graphs . . . 63
- 5.2 Simple BPaxos . . . 68
- 5.3 Fast Paxos . . . 77

5.4 Fast BPaxos 81
5.5 Tension Avoidance 85
5.6 Tension Resolution 90
5.7 Related Work 95
5.8 Conclusion 97

6 Matchmaker Paxos **101**
6.1 Matchmaker Paxos 103
6.2 Matchmaker MultiPaxos 113
6.3 Reconfiguring Matchmakers 118
6.4 Theoretical Insights 121
6.5 Evaluation 128
6.6 Related Work 133

7 Conclusion and Lessons Learned **137**

Bibliography **139**

List of Figures

2.1 An example execution of Paxos ($f = 1$). 5
2.2 At time $t = 0$, no state machine commands are chosen. At time $t = 1$ command x is chosen in slot 0. At times $t = 2$ and $t = 3$, commands z and y are chosen in slots 2 and 1. Executed commands are shaded green. Note that all state machines execute the commands x, y, z in log order. 8
2.3 An example execution of MultiPaxos ($f = 1$). The leader is adorned with a crown. 9

3.1 An example execution of Compartmentalized MultiPaxos with three proxy leaders ($f = 1$). Throughout the chapter, nodes and messages that were not present in previous iterations of the protocol are highlighted in green. 13
3.2 An execution of Compartmentalized MultiPaxos with a 2×3 grid of acceptors ($f = 1$). The two read quorums—$\{a_1, a_2, a_3\}$ and $\{a_4, a_5, a_6\}$—are shown in solid red rectangles. The three write quorums—$\{a_1, a_4\}$, $\{a_2, a_5\}$, and $\{a_3, a_6\}$—are shown in dashed blue rectangles. 16
3.3 An example execution of Compartmentalized MultiPaxos with three replicas as opposed to the minimum required two ($f = 1$). 17
3.4 An example execution of Compartmentalized MultiPaxos' read and write path ($f = 1$) with a 2×2 acceptor grid. The write path is shown using solid blue lines. The read path is shown using red dashed lines. 19
3.5 20
3.6 An execution that is not linearizable 21
3.7 A history, H_{pending}, with a pending invocation 21
3.8 $H_{wwr} \mid c_1$ 22
3.9 H_{wrw} 22
3.10 S_{wrw} 23
3.11 A motivating example of history extension 23
3.12 An example history G. Responses are not shown, as they are not important for this example. 24
3.13 A linearization S_G of the history in G Figure 3.12 24

3.14 An example execution of Compartmentalized MultiPaxos with batching ($f = 1$). Messages that contain a batch of commands, rather than a single command, are drawn thicker. Note how replica r_2 has to send multiple messages after executing a batch of commands. 27
3.15 An example execution of Compartmentalized MultiPaxos with batchers ($f = 1$). 28
3.16 An example execution of Compartmentalized MultiPaxos with unbatchers ($f = 1$). 29
3.17 A Mencius log round robin partitioned among three leaders. 30
3.18 An example of using noops to deal with a slow leader. Leader l_3 is slower than leaders l_1 and l_2, so the log has holes in l_3's slots. l_3 fills its holes with noops to allow commands in the log to be executed. 31
3.19 An example execution of Mencius. 31
3.20 An example execution of decoupled Mencius. Note that every proposer is a leader. 32
3.21 An execution of Mencius with proxy leaders, acceptor grids, and an increased number of replicas. 33
3.22 An example execution of S-Paxos. Messages that include client commands (as opposed to ids) are bolded. 34
3.23 An example execution of decoupled S-Paxos. Messages that include client commands (as opposed to ids) are bolded. Note that the MultiPaxos leader does not send or receive any messages that include a command, only messages that include command ids. 35
3.24 An example execution of S-Paxos with stabilizer grids, proxy leaders, acceptor grids, and an increased number of replicas. Messages that include client commands (as opposed to ids) are bolded. 36
3.25 The latency and throughput of MultiPaxos, Compartmentalized MultiPaxos, and an unreplicated state machine. 37
3.26 The latency and throughput of Compartmentalized MultiPaxos and an unreplicated state machine without batching and with larger value sizes. 39
3.27 An ablation study. Standard deviations are shown using error bars. 39
3.28 Peak throughput vs the number of replicas 40
3.29 Analytical throughput vs the number of replicas. 42
3.30 Peak throughput vs the number of replicas 43
3.31 The effect of skew on Compartmentalized MultiPaxos and CRAQ. 44

4.1 The 2 by 3 grid quorum system $Q_{2\times 3}$. 50
4.2 Quorum systems, capacity, and fault tolerance. 52
4.3 Heterogeneous nodes with different capacities. 53
4.4 A distribution of read fractions. 54
4.5 Strategy capacities with respect to read fraction 54
4.6 0-resilient and 1-resilient strategies. 56
4.7 Finding a latency-optimal strategy with capacity and network load constraints. 57
4.8 Searching the space of quorum systems. 58

4.9 Nodes and workload distribution. 58
4.10 The read quorums of the staggered grid and paths quorum systems. The optimal set of complementary write quorums is chosen automatically. 59
4.11 Quorum systems and their capacities. 59
4.12 Searching for a load-optimal quorum system. 59
4.13 Latencies with a capacity constraint. 60
4.14 A stacked histogram of the throughput of a simple majority quorum system with a naive uniform strategy. Write quorums are in blue, and read quorums are in red. 61
4.15 A stacked histogram of the throughput of the quorum system found by our heuristic search (i.e., the quorum system with read quorums $(c + bd)(a + e)$). 61

5.1 If two commands commute, replicas can safely execute them in either order. . . 64
5.2 A log and corresponding conflict graph. 65
5.3 In subfigures (a) - (e), we see a conflict graph constructed over time. The most recently chosen vertex is drawn in red. The executed commands are shaded green. (a) The command `a=b` is chosen in vertex v_0 without any dependencies. The command is executed immediately. (b) The command `a=2` is chosen in vertex v_1 with a dependency on v_0. The command is executed immediately. (c) The command `b=a` is chosen in vertex v_3 with dependencies on v_0, v_1, v_2, and v_4. No commands have been chosen in v_2 and v_4 yet, so v_3 cannot be executed. (d) The command `b=1` is chosen in vertex v_2 with a dependency on v_0. The command is executed immediately. (e) The command `a=3` is chosen in vertex v_4 with dependencies on v_0, v_1, and v_3. Now v_3 and v_4 are executed. In subfigures (f) - (j), we see an analogous execution for a log. 66
5.4 An example execution of Simple BPaxos ($f = 1$). 68
5.5 In subfigures (a) – (e), we see the execution of a dependency service node d_i. (a) d_i receives command w in vertex v_w. d_i adds this vertex to its conflict graph and because there are no other vertices, it returns the dependencies $\text{deps}(v_w) = \emptyset$. (b) d_i receives command x in vertex v_x. d_i adds this vertex to its conflict graph. x conflicts with w, so d_i adds an edge from v_x to v_w and returns the dependencies $\text{deps}(v_x) = \{v_w\}$. (c) d_i receives command y in vertex v_y. d_i adds this vertex to its conflict graph. y conflicts with w and x, so d_i adds an edge from v_y to v_w and from v_y to v_x. It returns the dependencies $\text{deps}(v_y) = \{v_w, v_x\}$. (d) d_i receives command z in vertex v_z. d_i adds this vertex to its conflict graph. z conflicts with w and x, so d_i adds an edge from v_z to v_w and from v_z to v_x. It returns the dependencies $\text{deps}(v_z) = \{v_w, v_x\}$. (e) d_i receives command x in vertex v_x. d_i's graph already contains vertex v_x, so d_i returns the dependencies $\text{deps}(v_x) = \{v_w\}$ and does not modify its graph. 70
5.6 An example execution of Simple BPaxos ($f = 1$). 72
5.7 An example execution of Simple BPaxos recovery ($f = 1$). 73
5.8 An example of dependency compaction 76

5.9 Example executions of Fast Paxos ($f = 2$). The leader of round 0 is adorned with a crown. Client requests are drawn in red. Phase 1 messages are drawn in blue. Phase 2 messages are drawn in green. 79
5.10 An example execution of Fast BPaxos ($f = 1$). The four network delays are drawn in red. 82
5.11 A Fast BPaxos bug ($f = 2$). Conflicting commands x and y are executed in different orders by different replicas. 84
5.12 An example execution of Unanimous BPaxos ($f = 2$) with the Basic EPaxos optimization. The messages introduced by the optimization are drawn in red. . 88
5.13 A non-exhaustive taxonomy of state machine replication protocols. The generalized multi-leader protocols that we discuss in this chapter are shaded green. 96

6.1 Matchmaker Paxos ($f = 1$). 104
6.2 A matchmaker's log over time. (a) Initially, the matchmaker's log is empty. (b) Then, the matchmaker receives MatchA$\langle 0, C_0 \rangle$. It inserts C_0 in log entry 0 and returns MatchB$\langle 0, \emptyset \rangle$ since the log does not contain any configuration in any round less than 0. (c) The matchmaker then receives MatchA$\langle 2, C_2 \rangle$. It inserts C_2 in log entry 2 and returns MatchB$\langle 2, \{(0, C_0)\} \rangle$. (d) It then receives MatchA$\langle 3, C_3 \rangle$, inserts C_3 in log entry 3, and returns MatchB$\langle 3, \{(0, C_0), (2, C_2)\} \rangle$. At this point, if the matchmaker were to receive MatchA$\langle 1, C_1 \rangle$, it would ignore it. 105
6.3 A leader's knowledge of the log after Phase 1. 113
6.4 An example Matchmaker MultiPaxos reconfiguration without Phase 1 bypassing. The leader p_1 reconfigures from the acceptors a_1, a_2, a_3 to the acceptors b_1, b_2, b_3. Client commands are drawn as gray dashed lines. Note that every subfigure shows one phase of a reconfiguration using solid colored lines, but the dashed lines show the complete execution of a client request that runs concurrently with the reconfiguration. For simplicity, we assume that every proposer also serves as a replica. 115
6.5 An example of merging three matchmaker logs (L_0, L_1, and L_2) during a matchmaker reconfiguration. Garbage collected log entries are shown in red. 119
6.6 A MultiPaxos log during reconfiguration ($\alpha = 4$). 121
6.7 Matchmaker MultiPaxos' latency and throughput ($f = 1$). Median latency is shown using solid lines, while the 95% latency is shown as a shaded region above the median latency. The vertical black lines show reconfigurations. The vertical dashed red line shows an acceptor failure. 129
6.8 The latency and throughput of MultiPaxos with horizontal reconfiguration ($f = 1$). 130
6.9 Violin plots of Figure 6.7 latency and throughput during the first 10 seconds and between 10 and 20 seconds. 131

6.10 Violin plots of Figure 6.8 latency and throughput during the first 10 seconds and between 10 and 20 seconds. 132
6.11 Matchmaker MultiPaxos' latency and throughput ($f = 1$). The dashed red line denotes a leader failure. 134
6.12 The latency and throughput of Horizontal MultiPaxos with $f = 1$. 135
6.13 The latency and throughput of Matchmaker MultiPaxos ($f = 1$). The dotted blue, dashed red, and vertical black lines show matchmaker reconfigurations, a matchmaker failure, and an acceptor reconfiguration respectively. 136

List of Tables

3.1 A summary of the compartmentalizations presented in this chapter. 12

5.1 The differences between protocols like MultiPaxos and Raft that organize commands in logs and protocols like EPaxos, Caesar, and Atlas that organize commands in graphs. 67
5.2 A summary of generalized multi-leader state machine replication protocols. . . 100

6.1 Figure 6.7 median, interquartile range, and standard deviation of latency and throughput. 133
6.2 Figure 6.13 median, interquartile range, and standard deviation of latency and throughput. 136

Acknowledgments

I am immensely grateful to Joseph Hellerstein, my advisor. Joe, you are one of the biggest reasons I came to Berkeley, and I am so happy I did. You know so much about so much, and you grok things so quickly. You have always been in my corner and have always had my back. You put your full faith and confidence in me and motivated me throughout my PhD. You were a mentor and a friend to whom I could always turn. From the bottom of my heart, thank you Joe!

I was so fortunate to work closely with Ion Stoica. Ion, you have always been so sharp, always able to predict what I'm going to say before I say it. I've benefited tremendously from your experience and insights. From paper advice to career advice to life advice, I always relish the opportunity to learn from you. Thank you Ion!

Natacha Crooks, we missed each other at Cornell and overlapped for only a year during a quarantine at Berkeley. I really wish we had more overlap because collaborating with you has been amazing. You are such a friendly and uplifting mentor, and I always felt like you were looking out for me. I hope we cross paths again in the future.

Barna Saha, thank you for serving on my qual and dissertation committee. Your feedback is always insightful.

Neil Giridharan, we've been working together since the day I started working on consensus. You have been an absolute dream of a collaborator, and I can't imagine doing any of the research without you. Three years ago, we used to talk about vanilla Paxos, and now you're inventing Byzantine consensus protocols with the world's leading experts. I am so proud of what you have already achieved and will continue to achieve, and I wish you the best of luck in grad school!

Heidi Howard, you are in my opinion the world's leading expert on consensus and state machine replication. Heidi, your effect on me started before we even met. Your work on Raft in OCaml, your conversations with Diego online, and your videos on Paxos on YouTube all encouraged me to research state machine replication. It was an honor being able to meet and work with you. Your input has been absolutely critical to most of the work I've done during grad school. You led me towards optimizing reads in Compartmentalized Paxos, you guided Quoracle to becoming a fully fleshed idea, you gave valuable insights into Matchmaker Paxos, and so on. Thank you Heidi!

Thank you to the trio of Aleksey Charapko, Murat Demirbas, and Eddie Ailijiang. Your work on Paxos quorum reads is a huge component of Compartmentalized Paxos, and PigPaxos has a major influence on Compartmentalized Paxos as well. Working together has been an absolute joy, and attending your reading group was always the highlight of my week. Aleksey, we also went on to work on Quoracle together, which was amazing. I cherish every opportunity we have to work together. Your knowledge on Paxos is unparalleled, and you've been a big inspiration for me.

I had the opportunity to collaborate with Irene Zhang, Dan Ports, Adriana Szekeres, Naveen Sharma, and Jialin Li at the University of Washington on TAPIR and Meerkat. You all welcomed me into your lab with such sincerity, and I learned so much from working with

all of you. Adriana, my friend, we went on to collaborate on most of the work I did during my PhD. You are one of the most positive and friendliest people I have ever met, and you were such a good mentor to me for so many years. Thank you so much!

Faisal Nawab, I remember sending you a question about one of your papers over email, and you helped grow that into an amazing collaboration and research project. You visited Berkeley to give a talk and we went on a small walk. Even though that was the first time I had spent any real time with you in person, you were so friendly, it felt like we had been friends for years. I'm very happy we had the chance to work together.

Peter Alvaro, my academic older brother, I remember watching your job talk when I was still an undergraduate at Cornell and being blown away. I was so lucky to have worked with you as a young grad student. You were a role model in how to conduct research, how to give good talks, and how to be an overall good dude. I still remember my first talk as a grad student was at SOCC 2018, and we sat in your hotel room, skipping other talks, as you helped get ready for my talk. Thank you Peter!

In the summer of 2018, I interned at Google with an amazing group of colleagues. Ultimately, I published a paper on my intern project with Nick Edmonds, Sandeep Tata, James B. Wendt, and Marc Najork. This paper was a major team effort and would never have happened without everyone's contributions. In particular, Nick, you put in a tremendous amount of work taking my toy intern project and making it production ready. We wouldn't have even come close to a paper without your enormous contributions. And Sandeep, you helped me at every step of the project. You taught me how to be a kind and effective leader that empowers everyone around them. Thank you to the Juicer team!

Nate Foster, my undergraduate advisor, I can say in a very literal sense that without you, I would not have made it to grad school. Even though I had absolutely no research experience at the time, you led me and a group of other undergraduates to publish within six months. You helped me at every single step of applying to grad school. You inspired me to pursue a PhD and continue to inspire me to help others. Thank you Nate!

I am so grateful for all of my labmates. My academic siblings, Charles, Chenggang, David, Johann, Larry, Rolando, Vikram, and Yifan, you have always been an amazing support system. Yifan, you are such a friendly and genuine person. I think I still owe you a hot pot. David and Kaushik, working with you on your research has been a blast. I'm confident the work is going to be very successful. Stephanie, I couldn't have asked for a better person to start the PhD with. You have done so much to bring the lab together. The Tahoe trip was simply amazing. Lisa, I always look forward to Friday board games. Thank you for organizing that and welcoming everyone. The lab is so lucky to have you. Peter, my neighbor and one time bunkmate, I'll miss sitting next to you. Eyal, keep representing Cornell proudly! You are such a chill dude, and I always enjoyed staying in lab late with you. Paras, I'm happy our paths crossed again after high school. Zongheng, when I first met you, I thought that you were a genius and that you were so intimidating. As I got to know you and learn how friendly of a person you are, I realized that only one of those things is true. Jeongseok, I enjoyed our time at the Monterey Bay Aquarium together, and I'm excited for our paths to cross in industry. And thank you to all the other labmates who influenced

me throughout grad school: Alexey, Amir, Anand, Ankur, Anurag, Audrey, Ben, Bobby, Dan, Daniel, Devin, Doris, Doris, Eric, Frank, Gabe, Ionel, Jean-Luc, Joao, Jose, Justin, Lily, Matthew, Melih, Michael, Milano, Moustafa, Nathan, Nilesh, Philipp, Qifan, Richard, Rishabh, Robert, Romil, Sagar, Sam, Samvit, Samy, Samyu, Sanjay, Sarah, Stephen, Sukrit, Vinamra, Vlad, Wenting, and many others.

A huge thank you to the RISE staff, Boban, Dave, Jon, Kattt, and Shane. You are the backbone of the lab, and I would have been lost without all of your help.

Finally, my partner Ava, you have been my rock. Without you, I definitely would not have made it through grad school. Ava, I love you and appreciate all that you do for me. You've listened to me talk about Paxos for three years straight and haven't left me yet, so it must be true love.

Chapter 1

Introduction

Imagine deploying a service—like a key-value store, for example—on a single server. If that server fails, then the service becomes completely unavailable, and all of the data stored in the system becomes irreparably lost. To handle failures more gracefully, we deploy services as distributed systems that are replicated redundantly on multiple servers. Rather than deploying one copy of a key-value store on one server, we deploy multiple copies, or replicas, on multiple servers. But, to avoid anomalous behavior, we must make sure that the replicas stay in sync. State machine replication is the de facto solution to this problem [78]. State machine replication protocols model services, like a key-value store, as deterministic state machines and ensure that every replica of the state machine is always in sync.

Thus, state machine replication protocols—like MultiPaxos [45, 87] and Raft [68]—are crucial components of almost every strongly consistent distributed system and database [18, 86, 11, 83, 27, 1, 72, 52]. However, despite their importance, these protocols are often viewed as a necessary evil that should be avoided if possible. People think these protocols are too slow [62, 97], too complicated [87, 68, 59], and too hard to implement [12, 54]. This thesis tackles many aspects of consensus and state machine replication, showing that these protocols do not have to be slow, complicated, or hard to implement. The thesis focuses on four main pieces of work: Compartmentalized Paxos, Quoracle, Bipartisan Paxos, and Matchmaker Paxos.

Compartmentalized Paxos

MultiPaxos is notoriously slow [44, 87, 68]. Ideally, we could scale up the protocol to increase its throughput. However, state machine replication is fundamentally sequential and there are no immediately obvious opportunities for parallelization or scaling. In fact, naively scaling up MultiPaxos decreases the protocol's throughput. We introduce a design principle and optimization technique called **compartmentalization** and apply it to MultiPaxos to significantly increase its throughput. The main idea behind compartmentalization is to first *decouple* protocol components into subcomponents and then independently *scale* up the subcomponents. Through this decoupling, we disentangle the components of state machine

replication that are fundamentally sequential from those that are embarrassingly parallel. This enables us to scale up a majority of the protocol while isolating the fundamentally unscalable components.

Quoracle

Imagine we have n replicas of a piece of state and that clients write to w replicas and read from r replicas. If $r + w > n$, then the set of replicas from which a client reads overlaps the set of replicas to which a client writes, so clients are guaranteed to read the latest write. This idea of overlapping sets of machines, formalized as **quorum systems**, are a key part of almost every state machine replication protocol and become even more important once we introduce compartmentalization. The quorum system a protocol uses can have a big impact on its performance. As a result, researchers have invented a lot of quorum systems [70, 2, 26, 24, 15, 56, 39, 16, 64]. There is a body of theory that underlies and subsumes these quorum systems [64, 34], but this theory is inaccessible for a number of reasons. First, it is dense and relatively unknown. Second, it is too theoretical and ignores many practical considerations. Third, there is no tooling to apply the theory in practice. We address all three shortcomings. We revisit and refine the theory with a number of practical considerations (e.g. latency, network load, heterogeneous deployments) that cannot be overlooked in a real-world deployment. We also implement the refined theory in a Python utility called **Quoracle** that can automatically construct, analyze, optimize, and select quorum systems using cost-based optimization. Industry practitioners can use this utility to optimize their distributed systems. The tool makes a large body of theory accessible to practitioners for the first time.

Bipartisan Paxos

We can compartmentalize a protocol to make it faster, but we can also compartmentalize a protocol to make it easier to understand. There is a family of state machine replication protocols called generalized multi-leader protocols [62, 21, 6]. The protocols deploy multiple leaders to avoid being bottlenecked by a single leader, and they maintain a partial order of state machine commands rather than a total order. These protocols are extremely complicated. Through personal conversations, we found that even experts on consensus admit to not understanding these protocols fully. As a result, all of these protocols start to look equally impenetrable. It is hard to answer questions like “how are the protocols different?”, “what are the strengths and weaknesses of the protocols?”, and “which areas of the trade-off space are still unexplored?” We use compartmentalization to dissect this class of protocols into its simplest components and understand each component in isolation. We then put the pieces back together in various ways to construct the existing protocols in the space. In doing so, we present a comprehensive architectural breakdown on this class of protocols. We also discover some interesting and counterintuitive takeaways along the way. For example, some protocols introduce sophisticated mechanisms to eliminate throughput bottlenecks, but af-

ter compartmentalizing other aspects of these protocols, these mechanisms become the very bottlenecks they try to eliminate.

Matchmaker Paxos

Machines fail, and when they do, they need to be replaced. Because state machine replication protocols are long running processes, the failed machines running the protocol have to be replaced as the protocol runs, a process known as **reconfiguration**. We propose a novel reconfiguration protocol called **Matchmaker Paxos** that is based on Vertical Paxos [49]. We introduce a new set of nodes called matchmakers that act as a source of truth for the current configuration. Nodes contact the matchmakers to learn about prior configurations and to register the current configuration. Reconfiguration is an absolutely essential part of state machine replication, but despite this, there is very little research on reconfiguration. Our protocol takes a step towards filling this void. Matchmaker Paxos is simple and fully proven, something that is surprisingly rare in the space. It also leads to a number of novel theoretical insights and can be used by many replication protocols that do not have a reconfiguration protocol.

Previous Publications

Compartmentalized Paxos was previously published in VLDB 2021 [95]. Quoracle was previously published in PaPoC 2021 [94]. At the time of writing, Bipartisan Paxos and Matchmaker Paxos were both under submission to JSys 2021. My previous work on invariant confluence [91, 90], wat-provenance [92], and Juicer [93] are not included in this dissertation.

Chapter 2

Background

In this chapter, we survey all of the background material that is needed for the remainder of the thesis. We start by clarifying the system model and assumptions we use throughout the thesis. We then introduce Single Decree Paxos [45], MultiPaxos [44], and Flexible Paxos [31]. This family of state machine replication protocols is arguably the oldest, most popular, and most widely used. We design a number of Paxos, MultiPaxos, and Flexible Paxos variants throughout the paper.

2.1 System Model

Throughout the thesis, we assume an asynchronous network model in which messages can be arbitrarily dropped, delayed, and reordered. We assume machines can fail by crashing but do not act maliciously. We assume that machines operate at arbitrary speeds, and we do not assume clock synchronization. Every protocol discussed in this thesis assumes (for liveness) that at most f machines will fail for some configurable f. All the protocols discussed in this thesis are safe, but due to the FLP impossibility result [23], none of the protocols are guaranteed to be fully live in a fully asynchronous network. Liveness is only guaranteed given partial synchrony.

2.2 Paxos

A **consensus protocol** is a protocol that selects a single value from a set of proposed values. The protocol must select a value that was proposed (non-triviality) and can only select a single value (consistency) [42]. **Single Decree Paxos** [45, 44], abbreviated Paxos, is one of the oldest and most popular consensus protocols. A Paxos deployment that tolerates f faults consists of an arbitrary number of clients, at least $f+1$ nodes called **proposers**, and $2f+1$ nodes called **acceptors**, as illustrated in Figure 2.1.[1] To reach consensus on a value, an

[1] It is also common for Paxos to include a set of **learners**, nodes that learn which value is chosen. In this thesis, we omit this role.

execution of Paxos is divided into a number of rounds (also known as ballots, epochs, terms, views, etc. [33]). Every round has two phases, Phase 1 and Phase 2, and is orchestrated by a single pre-determined proposer. The set of rounds can be any unbounded, totally ordered set (e.g., the set of natural numbers). In practice, it is common to let the set of rounds be the set of lexicographically ordered integer pairs (r, id) where r is an integer and id is a unique proposer id, where a proposer is responsible for executing every round that contains its id. For simplicity though, we often use natural numbers as rounds throughout the thesis.

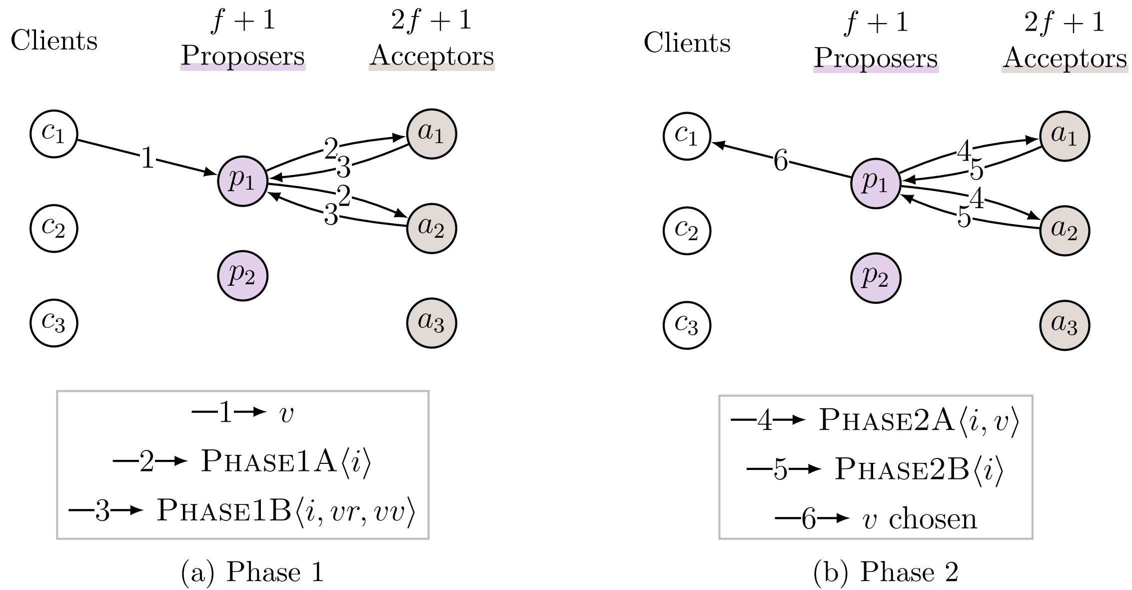


Figure 2.1: An example execution of Paxos ($f = 1$).

When a proposer executes a round, say round i, it attempts to get some value x chosen in that round. Paxos is a consensus protocol, so it must only choose a single value. Thus, Paxos must ensure that if a value x is chosen in round i, then no other value besides x can ever be chosen in any round less than i. This is the purpose of Paxos' two phases. In Phase 1 of round i, the proposer contacts the acceptors to (a) learn of any value that may have already been chosen in any round less than i and (b) prevent any new values from being chosen in any round less than i. In Phase 2, the proposer proposes a value to the acceptors, and the acceptors vote on whether or not to choose it. In Phase 2, the proposer will only propose a value x if it learned through Phase 1 that no other value has been or will be chosen in a previous round.

More concretely, Paxos executes as follows, as illustrated in Figure 2.1. When a client wants to propose a value x, it sends x to a proposer p. Upon receiving x, p begins executing one round of Paxos, say round i. First, it executes Phase 1. It sends $\textsc{Phase1A}\langle i \rangle$ messages to the acceptors. An acceptor ignores a $\textsc{Phase1A}\langle i \rangle$ message if it has already received a message in a larger round. Otherwise, it replies with a $\textsc{Phase1B}\langle i, vr, vv \rangle$ message con-

taining the largest round vr in which the acceptor voted and the value it voted for, vv. If the acceptor hasn't voted yet, then $vr = -1$ and $vv = \textsf{null}$. When the proposer receives PHASE1B messages from a majority of the acceptors, Phase 1 ends and Phase 2 begins.

At the start of Phase 2, the proposer uses the PHASE1B messages that it received in Phase 1 to select a value x such that no value other than x has been or will be chosen in any round less than i. Specifically x is the vote value associated with the largest received vote round, or any value if no acceptor had voted (see [44] for details). Then, the proposer sends PHASE2A$\langle i, x\rangle$ messages to the acceptors. An acceptor ignores a PHASE2A$\langle i, x\rangle$ message if it has already received a message in a larger round. Otherwise, it votes for x and sends back a PHASE2B$\langle i\rangle$ message to the proposer. If a majority of acceptors vote for the value, then the value is **chosen**, and the proposer informs the client.

If the proposer does not receive sufficiently many PHASE1B or PHASE2B responses from the acceptors (e.g., because of network partitions or dueling proposers), then the proposer restarts the protocol in a larger round.

Note that it is safe for the leader of round 0 (the smallest round) to skip Phase 1 and proceed directly to Phase 2. Recall that the leader of round i executes Phase 1 to learn of any value that may have already been chosen in any round less than i and to prevent any new values from being chosen in any round less than i. There are no rounds less than 0, so these properties are satisfied vacuously.

Safety Proof

Algorithm 1 Paxos Proposer Pseudocode

```
State: a value v, initially null
State: a round i, initially −1
 1: upon receiving a value x from a client do
 2:     i ← the next largest round owned by this proposer
 3:     v ← x
 4:     Send PHASE1A⟨i⟩ messages to the acceptors
 5: upon receiving PHASE1B⟨i, vr, vv⟩ messages from f + 1 acceptors do
 6:     k ← the largest vr in any PHASE1B⟨i, vr, vv⟩
 7:     if k ≠ −1 then
 8:         v ← the corresponding value vv in round k
 9:     send PHASE2A⟨i, v⟩ messages to the acceptors
10: upon receiving PHASE2B⟨i⟩ messages from f + 1 acceptors do
11:     v is chosen, notify the client
```

Paxos proposer and Paxos acceptor pseudocode is given in Algorithm 1 and Algorithm 2 respectively. For brevity, we omit details surrounding resending messages and retrying proposals in larger rounds. We now prove that Paxos is safe, using the proof from [41].

Algorithm 2 Paxos Acceptor Pseudocode

State: the largest seen round r, initially -1
State: the largest round vr voted in, initially -1
State: the value vv voted for in round vr, initially null

```
1: upon receiving a Phase1A⟨i⟩ message from proposer p with i > r do
2:     r ← i
3:     send a Phase1B⟨i, vr, vv⟩ message to p
4: upon receiving a Phase2A⟨i, x⟩ message from proposer p with i ≥ r do
5:     r, vr, vv ← i, i, x
6:     send a Phase2B⟨i⟩ message to p
```

Proof. We prove, for every round i, the statement $P(i)$: "if a proposer proposes a value v in round i (i.e. sends a Phase2A message for value v in round i), then no value other than v has been or will be chosen in any round less than i." At most one value is ever proposed in a given round, so at most one value is ever chosen in a given round. Thus, $P(i)$ suffices to prove that Paxos is safe for the following reason. Assume for contradiction that Paxos chooses distinct values x and y in rounds j and i with $j < i$. Some proposer must have proposed y in round i, so $P(i)$ ensures us that no value other than y could have been chosen in round j. But, x was chosen, a contradiction.

We prove $P(i)$ by strong induction on i. $P(0)$ is vacuous because there are no rounds less than 0. For the general case $P(i)$, we assume $P(0), \ldots, P(i-1)$. We perform a case analysis on the proposer's pseudocode (Algorithm 1). Either k is -1 or it is not (line 6). First, assume it is not (line 7). In this case, the proposer proposes v, the value proposed in round k (line 8). We perform a case analysis on round j to show that no value other than v has been or will be chosen in any round $j < i$.

- **Case 1:** $j > k$**.** The proposer sent Phase1A$\langle i \rangle$ messages to all of the acceptors. $f+1$ of these acceptors, say Q, all received Phase1A$\langle i \rangle$ messages and replied with Phase1B messages. Thus, every acceptor in Q set its round r to i, and in doing so, promised to never vote in any round less than i. Moreover, none of the acceptors in Q had voted in any round greater than k. So, every acceptor in Q has not voted and never will vote in round j. For a value v' to be chosen in round j, it must receive votes from some set Q' of $f+1$ acceptors in round j. But, Q and Q' necessarily intersect, so this is impossible. Thus, no value has been or will be chosen in round j.

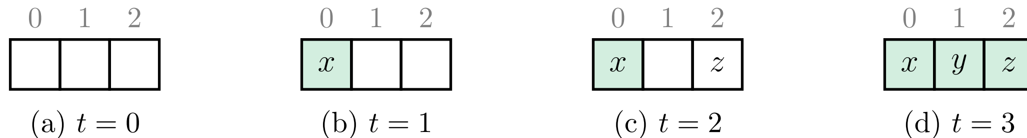
- **Case 2:** $j = k$**.** In a given round, at most one value is proposed, let alone chosen. v is *the* value proposed in round k, so no other value could be chosen in round k.

- **Case 3:** $j < k$**.** By induction, $P(k)$ states that no value other than v has been or will be chosen in any round less than k. This includes round j.

Finally, if k is -1, then we are in the same situation as in Case 1. No value has or will be chosen in a round $j < i$, so the proposer is free to propose any value. Specifically, it can propose the value x it received from the client (line 3). □

2.3 MultiPaxos

While consensus is the act of choosing a single value, **state machine replication** is the act of choosing a sequence (a.k.a. log) of values [78]. A state machine replication protocol manages a number of **replicas** of a deterministic state machine. Over time, the protocol constructs a growing log of state machine commands, and replicas execute the commands in log order. By beginning in the same initial state, and by executing commands in the same order, all state machine replicas are kept in sync. This is illustrated in Figure 2.2.



(a) $t = 0$ (b) $t = 1$ (c) $t = 2$ (d) $t = 3$

Figure 2.2: At time $t = 0$, no state machine commands are chosen. At time $t = 1$ command x is chosen in slot 0. At times $t = 2$ and $t = 3$, commands z and y are chosen in slots 2 and 1. Executed commands are shaded green. Note that all state machines execute the commands x, y, z in log order.

MultiPaxos [45, 87] is one of the most widely used state machine replication protocols. MultiPaxos uses one instance of Paxos for every log entry, choosing the command in the ith log entry using the ith instance of Paxos. A MultiPaxos deployment that tolerates f faults consists of an arbitrary number of clients, at least $f+1$ proposers, and $2f+1$ acceptors (like Paxos), as well as at least $f+1$ replicas, as illustrated in Figure 2.3.

Initially, one of the proposers is elected leader in round i and runs Phase 1 of Paxos for every log entry. When a client wants to propose a state machine command x, it sends the command to the leader (1). The leader assigns the command a log entry j and then runs Phase 2 of the jth Paxos instance to get the value x chosen in entry j. That is, the leader sends $\text{Phase2A}\langle i, x\rangle$ messages to the acceptors to vote for value x in slot j (2). In the normal case, the acceptors all vote for x in slot j and respond with $\text{Phase2B}\langle i\rangle$ messages (3). Once the leader learns that a command has been chosen in a given log entry (i.e. once the leader receives $\text{Phase2B}\langle i\rangle$ messages from a majority of the acceptors), it informs the replicas (4). Replicas insert commands into their logs and execute the logs in prefix order.

Note that the leader assigns log entries to commands in increasing order. The first received command is put in entry 0, the next command in entry 1, the next command in entry 2, and so on. Also note that even though every replica executes every command, for any given state machine command x, only one replica needs to send the result of executing

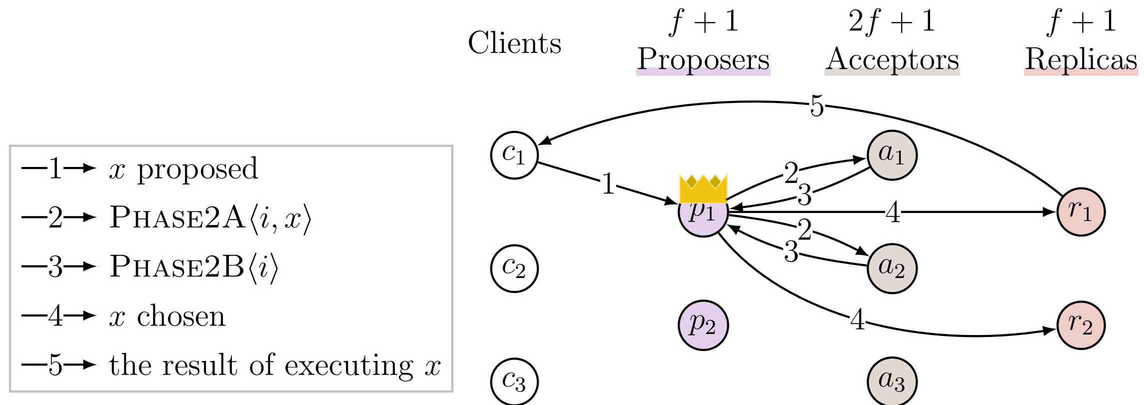


Figure 2.3: An example execution of MultiPaxos ($f = 1$). The leader is adorned with a crown.

x back to the client (5). For example, log entries can be round-robin partitioned across the replicas.

2.4 Flexible Paxos

Paxos deploys a set of $2f+1$ acceptors, and proposers communicate with at least a *majority* of the acceptors in Phase 1 and in Phase 2. While this is sufficient to ensure safety, it is not necessary. **Flexible Paxos** [31] is a Paxos variant that generalizes the notion of a *majority* to that of a *quorum*. Specifically, Flexible Paxos uses the notion of a read-write quorum system [64]. Given a set $A = \{a_1, \ldots, a_n\}$ of acceptors, a **read-write quorum system** over A is a pair $Q = (R, W)$ where

1. R is a set of subsets of A called **read quorums** (or **Phase 1 quorums**),
2. W is a set of subsets of W called **write quorums** (or **Phase 2 quorums**), and
3. every read quorum intersects every write quorum. That is, for every $r \in R$ and $w \in W$, $r \cap w \neq \emptyset$

Flexible Paxos is identical to Paxos with the exception that proposers now communicate with an arbitrary Phase 1 quorum in Phase 1 and an arbitrary Phase 2 quorum in Phase 2. This suffices for safety. Flexible MultiPaxos is identical to MultiPaxos, except that it uses Flexible Paxos instead of Paxos. Note that in order for a configuration to tolerate f failures, there must exist some Phase 1 quorum and some Phase 2 quorum of non-failed machines despite an arbitrary set of f failures.

Chapter 3

Compartmentalization

In many state machine replication protocols, a single node has multiple responsibilities. For example, a Raft [68] leader acts as a batcher, a sequencer, a broadcaster, *and* a state machine replica. These overloaded nodes are often a throughput bottleneck, which can be disastrous for systems that rely on state machine replication.

Many databases, for example, rely on state machine replication to replicate large data partitions of tens of gigabytes [79, 27]. These databases require high-throughput state machine replication to handle all the requests in a partition. However, in such systems, it is not uncommon to exceed the throughput budget of a partition. For example, Cosmos DB will split a partition if it experiences high throughput even if the paritition is under-utilizing memory and disk. The split, aside from costing resources, may have additional adverse effects on applications, as Cosmos DB provides strongly consistent transactions only within the partition. Eliminating state machine replication bottlenecks can help avoid such unnecessary partition splits and improve performance, consistency, and resource utilization.

Researchers have studied how to eliminate throughput bottlenecks, often by inventing new state machine replication protocols that eliminate a *single* throughput bottleneck [62, 6, 57, 4, 42, 41, 13, 32, 98, 85, 10]. However, eliminating a *single* bottleneck is not enough to achieve the best possible throughput. When you eliminate one bottleneck, another arises. To achieve the best possible throughput, we have to repeatedly eliminate bottlenecks as they arise.

The key to eliminating these throughput bottlenecks is scaling, but it is widely believed that state machine replication protocols do not scale [37, 97, 57, 62, 6]. In this chapter, we show that this is not true. State machine replication protocols can indeed scale. As a concrete illustration, we analyze the throughput bottlenecks of MultiPaxos [44] and systematically eliminate them using a combination of decoupling and scaling, a technique we call **compartmentalization**. For example, consider the MultiPaxos leader, a notorious throughput bottleneck. The leader has two distinct responsibilities. First, it sequences state machine commands into a log. It puts the first command it receives into the first log entry, the next command into the second log entry, and so on. Second, it broadcasts the commands to the set of MultiPaxos acceptors, receives their responses, and then broadcasts the com-

mands again to a set of state machine replicas. To compartmentalize the MultiPaxos leader, we first **decouple** these two responsibilities. There is no fundamental reason that the leader has to sequence commands *and* broadcast them. Instead, we have the leader sequence commands and introduce a new set of nodes, called proxy leaders, to broadcast the commands. Second, we **scale** up the number of proxy leaders. We note that broadcasting commands is embarrassingly parallel, so we can increase the number of proxy leaders to avoid them becoming a bottleneck. Note that this scaling was not possible when sequencing and broadcasting were coupled on the leader since sequencing is not scalable. Compartmentalization has three key strengths.

(1) Fast Without Strong Assumptions. We compartmentalize MultiPaxos and increase its throughput by over an order of magnitude. Moreover, we achieve our strong performance without the strong assumptions made by other state machine replication protocols with comparable performance [85, 98, 88, 84, 35]. For example, we do not assume a perfect failure detector [88], we do not assume the availability of specialized hardware [98, 84, 35], we do not assume uniform data access patterns [85, 98], we do not assume clock synchrony [98], and we do not assume key-partitioned state machines [85, 98].

(2) General and Incrementally Adoptable. Compartmentalization is *not* a protocol. Rather, it is a technique that can be systematically applied to existing protocols. Industry practitioners can incrementally apply compartmentalization to their current protocols without having to throw out their battle-tested implementations for something new and untested. We demonstrate the generality of compartmentalization by applying it to three other protocols [57, 10, 20] in addition to MultiPaxos.

(3) Easy to Understand. Researchers have invented new state machine replication protocols to eliminate throughput bottlenecks, but these new protocols are often subtle and complicated. As a result, these sophisticated protocols have been largely ignored in industry due to their high barriers to adoption. Compartmentalization is based on the simple principles of decoupling and scaling and is designed to be easily understood. Moreover, in Chapter 5, we see how compartmentalization can be used to simplify existing protocols.

In this chapter, we introduce the technique of compartmentalization and use it to compartmentalize MultiPaxos, Mencius [57], and S-Paxos [10]. Later in Chapter 5, we compartmentalize more sophisticated state machine replication protocols like EPaxos [62].

3.1 MultiPaxos Does Not Scale?

It is widely believed that MultiPaxos does not scale [37, 97, 57, 62, 6]. Throughout this chapter, we will explain that this is not true, but it first helps to understand why trying to scale MultiPaxos in the straightforward and obvious way does not work. MultiPaxos consists of proposers, acceptors, and replicas. We discuss each.

First, increasing the number of proposers *does not improve performance* because every client must send its requests to the leader regardless of the number proposers. The non-leader proposers are idle and do not contribute to the protocol during normal operation.

Second, increasing the number of acceptors *hurts performance*. To get a value chosen, the leader must contact a majority of the acceptors. When we increase the number of acceptors, we increase the number of acceptors that the leader has to contact. This decreases throughput because the leader—which is the throughput bottleneck—has to send and receive more messages per command. Moreover, every acceptor processes at least half of all commands regardless of the number of acceptors.

Third, increasing the number of replicas *hurts performance*. The leader broadcasts chosen commands to all of the replicas, so when we increase the number of replicas, we increase the load on the leader and decrease MultiPaxos' throughput. Moreover, every replica must execute every state machine command, so increasing the number of replicas does not decrease the replicas' load.

3.2 Compartmentalizing MultiPaxos

We now compartmentalize MultiPaxos. Throughout this chapter, we introduce six compartmentalizations, summarized in Table 3.1. For every compartmentalization, we identify a throughput bottleneck and then explain how to decouple and scale it.

Table 3.1: A summary of the compartmentalizations presented in this chapter.

	Bottleneck	Decouple	Scale
1	leader	command sequencing and command broadcasting	the number of proxy leaders
2	acceptors	read quorums and write quorums	the number of write quorums
3	replicas	command sequencing and command broadcasting	the number of replicas
4	leader and replicas	read path and write path	the number of read quorums
5	leader	batch formation and batch sequencing	the number of batchers
6	replicas	batch processing and batch replying	the number of unbatchers

Compartmentalization 1: Proxy Leaders

Bottleneck: *leader*
Decouple: *command sequencing and broadcasting*
Scale: *the number of command broadcasters*

Bottleneck The MultiPaxos leader is a well known throughput bottleneck for the following reason. Refer again to Figure 2.3. To process a single state machine command from a client, the leader must receive a message from the client, send at least $f + 1$ PHASE2A messages to the acceptors, receive at least $f + 1$ PHASE2B messages from the acceptors, and send at least $f + 1$ messages to the replicas. In total, the leader sends and receives at least $3f + 4$ messages per command. Every acceptor on the other hand processes only 2 messages, and

every replica processes either 1 or 2. Because every state machine command goes through the leader, and because the leader has to perform disproportionately more work than every other component, the leader is the throughput bottleneck.

Decouple To alleviate this bottleneck, we first decouple the leader. To do so, we note that a MultiPaxos leader has two jobs. The first is **sequencing**. The leader sequences commands by assigning each command a log entry. Log entry 0, then 1, then 2, and so on. The second is **broadcasting**. The leader sends Phase2A messages, collects Phase2B responses, and broadcasts chosen values to the replicas. Historically, these two responsibilities have both fallen on the leader, but this is not fundamental. We instead decouple the two responsibilities. We introduce a set of at least $f + 1$ **proxy leaders**, as shown in Figure 3.1. The leader is responsible for sequencing commands, while the proxy leaders are responsible for getting commands chosen and broadcasting the commands to the replicas.

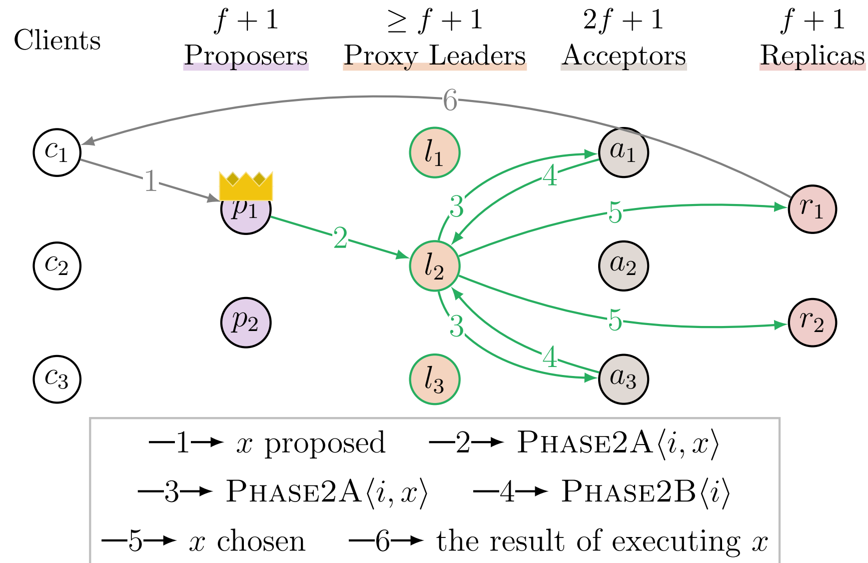


Figure 3.1: An example execution of Compartmentalized MultiPaxos with three proxy leaders ($f = 1$). Throughout the chapter, nodes and messages that were not present in previous iterations of the protocol are highlighted in green.

More concretely, when a leader receives a command x from a client (1), it assigns the command x a log entry i and then forms a Phase2A message that includes x and i. The leader does *not* send the Phase2A message to the acceptors. Instead, it sends the Phase2A message to a randomly selected proxy leader (2). Note that every command can be sent to a different proxy leader. The leader balances load evenly across all of the proxy leaders. Upon receiving a Phase2A message, a proxy leader broadcasts it to the acceptors (3), gathers a quorum of $f + 1$ Phase2B responses (4), and notifies the replicas of the chosen value (5). All other aspects of the protocol remain unchanged.

Without proxy leaders, the leader processes $3f + 4$ messages per command. With proxy leaders, the leader only processes 2. This makes the leader significantly less of a throughput bottleneck, or potentially eliminates it as the bottleneck entirely.

Scale The leader now processes fewer messages per command, but every proxy leader has to process $3f+4$ messages. Have we really eliminated the leader as a bottleneck, or have we just moved the bottleneck into the proxy leaders? To answer this, we note that the proxy leaders are embarrassingly parallel. They operate independently from one another. Moreover, the leader distributes load among the proxy leaders equally, so the load on any single proxy leader decreases as we increase the number of proxy leaders. Thus, we can trivially increase the number of proxy leaders until they are no longer a throughput bottleneck.

Discussion Note that decoupling *enables* scaling. As discussed in Section 3.1, we cannot naively increase the number of proposers. Without decoupling, the leader is both a sequencer and broadcaster, so we cannot increase the number of leaders to increase the number of broadcasters because doing so would lead to multiple sequencers, which is not permitted. Only by decoupling the two responsibilities can we scale one without scaling the other.

Also note that the protocol remains tolerant to f faults regardless of the number of machines. However, increasing the number of machines does decrease the expected time to f failures (this is true for every protocol that scales up the number of machines, not just our protocol). We believe that increasing throughput at the expense of a shorter time to f failures is well worth it in practice because failed machines can be replaced with new machines using a reconfiguration protocol [44, 68]. The time required to perform a reconfiguration is many orders of magnitude smaller than the mean time between failures.

Compartmentalization 2: Acceptor Grids

Bottleneck: *acceptors*
Decouple: *read quorums and write quorums*
Scale: *the number of write quorums*

Bottleneck After compartmentalizing the leader, it is possible that the acceptors are the throughput bottleneck. It is widely believed that acceptors do not scale: "using more than $2f + 1$ [acceptors] for f failures is possible but illogical because it requires a larger quorum size with no additional benefit" [97]. As explained in Section 3.1, there are two reasons why naively increasing the number of acceptors is ill-advised.

First, increasing the number of acceptors increases the number of messages that the leader has to send and receive. This increases the load on the leader, and since the leader is the throughput bottleneck, this decreases throughput. This argument no longer applies. With the introduction of proxy leaders, the leader no longer communicates with the acceptors.

Increasing the number of acceptors increases the load on every individual proxy leader, but the increased load will not make the proxy leaders a bottleneck because we can always scale them up.

Second, every command must be processed by a majority of the acceptors. Thus, even with a large number of acceptors, every acceptor must process at least half of all state machine commands. This argument still holds.

Decouple We compartmentalize the acceptors by using non-majority read-write quorum systems [31]. MultiPaxos—the vanilla version, not the compartmentalized version—requires $2f + 1$ acceptors, and the leader communicates with $f + 1$ acceptors in both Phase 1 and Phase 2 (a majority of the acceptors). As discussed in Section 2.4, this majority quorum system is safe but not necessary. By using other read-write quorum systems, read quorums do not have to intersect other read quorums, and write quorums do not have to intersect other write quorums.

By decoupling read quorums from write quorums, we can reduce the load on the acceptors by using a more efficient quorum system. Specifically, we arrange the acceptors into an $r \times w$ rectangular grid, where $r, w \geq f + 1$. This is called a **grid quorum system**. Every row forms a read quorum, and every column forms a write quorum (r stands for row and for read). That is, a leader contacts an arbitrary row of acceptors in Phase 1 and an arbitrary column of acceptors for every command in Phase 2. Every row intersects every column, so this is a valid set of quorums.

A 2×3 acceptor grid is illustrated in Figure 3.2. There are two read quorums (the rows $\{a_1, a_2, a_3\}$ and $\{a_4, a_5, a_6\}$) and three write quorums (the columns $\{a_1, a_4\}$, $\{a_2, a_5\}$, $\{a_3, a_6\}$). Because there are three write quorums, every acceptor only processes one third of all the commands. This is not possible with majority quorums because with majority quorums, every acceptor processes at least half of all the commands, regardless of the number of acceptors.

Scale With majority quorums, every acceptor has to process at least half of all state machines commands. With grid quorums, every acceptor only has to process $\frac{1}{w}$ of the state machine commands. Thus, we can increase w (i.e. increase the number of columns in the grid) to reduce the load on the acceptors and eliminate them as a throughput bottleneck.

Discussion Note that, like with proxy leaders, decoupling *enables* scaling. With majority quorums, read and write quorums are coupled, so we cannot increase the number of acceptors without also increasing the size of all quorums. Acceptor grids allow us to decouple the number of acceptors from the size of write quorums, allowing us to scale up the acceptors and decrease their load.

Also note that increasing the number of write quorums increases the size of read quorums which increases the number of acceptors that a leader has to contact in Phase 1. We believe

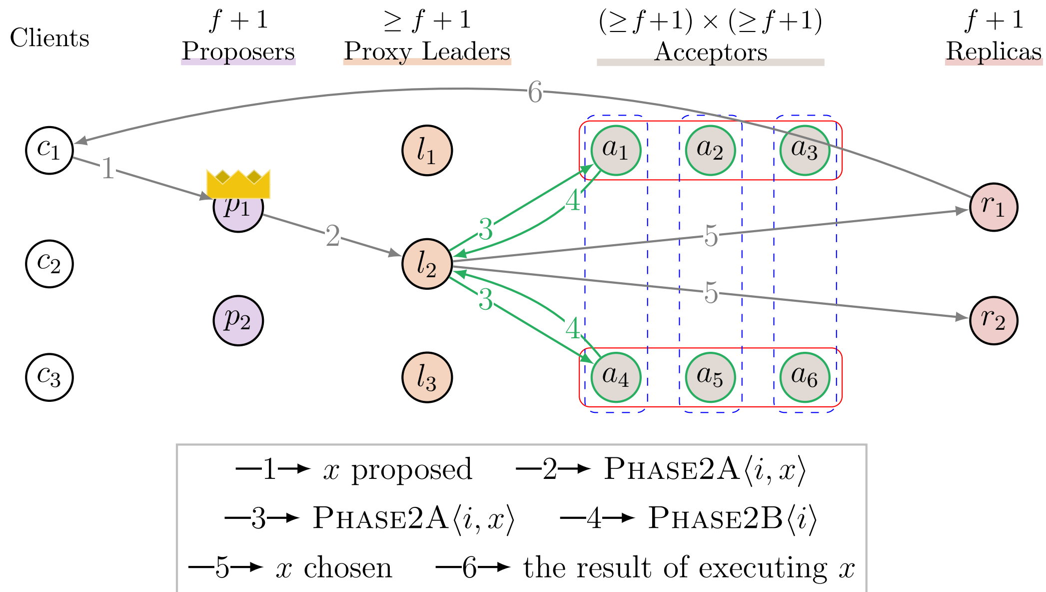


Figure 3.2: An execution of Compartmentalized MultiPaxos with a 2×3 grid of acceptors ($f = 1$). The two read quorums—$\{a_1, a_2, a_3\}$ and $\{a_4, a_5, a_6\}$—are shown in solid red rectangles. The three write quorums—$\{a_1, a_4\}$, $\{a_2, a_5\}$, and $\{a_3, a_6\}$—are shown in dashed blue rectangles.

this is a worthy trade-off since Phase 2 is executed in the normal case and Phase 1 is only run in the event of a leader failure.

In this chapter, we use the grid quorum system because it is simple to explain, but it is not always the most efficient quorum system. In Chapter 4, we take a deep dive into read-quorum systems. We can also use any of these sophisticated read-write quorum systems.

Compartmentalization 3: More Replicas

Bottleneck: *replicas*
Decouple: *command sequencing and broadcasting*
Scale: *the number of replicas*

Bottleneck After compartmentalizing the leader and the acceptors, it is possible that the replicas are the bottleneck. Recall from Section 3.1 that naively scaling the replicas does not work for two reasons. First, every replica must receive and execute every state machine command. This is not actually true, but we leave that for the next compartmentalization. Second, like with the acceptors, increasing the number of replicas increases the load on the leader. Because we have already decoupled sequencing from broadcasting on the leader and

introduced proxy leaders, this is no longer true, so we are free to increase the number of replicas. In Figure 3.3, for example, we show MultiPaxos with three replicas instead of the minimum required two.

Scale If every replica has to execute every command, does increasing the number of replicas decrease their load? Yes. Recall that while every replica has to execute every state machine, only *one* of the replicas has to send the result of executing the command back to the client. Thus, with n replicas, every replica only has to send back results for $\frac{1}{n}$ of the commands. If we scale up the number of replicas, we reduce the number of messages that each replica has to send. This reduces the load on the replicas and helps prevent them from becoming a throughput bottleneck. In Figure 3.3 for example, with three replicas, every replica only has to reply to one third of all commands. With two replicas, every replica has to reply to half of all commands. In the next compartmentalization, we'll see another major advantage of increasing the number of replicas.

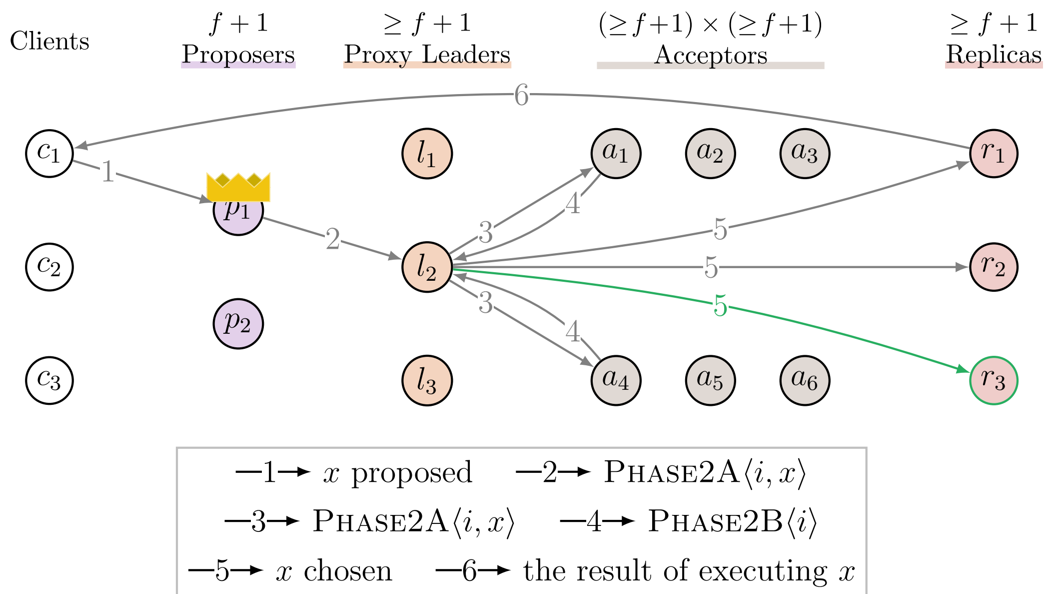


Figure 3.3: An example execution of Compartmentalized MultiPaxos with three replicas as opposed to the minimum required two ($f = 1$).

Discussion Again decoupling *enables* scaling. Without decoupling the leader and introducing proxy leaders, increasing the number of replicas hurts rather than helps performance.

Compartmentalization 4: Leaderless Reads

Bottleneck: *leader and replicas*
Decouple: *read path and write path*
Scale: *the number of read quorums*

Bottleneck We have now compartmentalized the leader, the acceptors, and the replicas. At this point, the bottleneck is in one of two places. Either the leader is still a bottleneck, or the replicas are the bottleneck. Fortunately, we can bypass both bottlenecks with a single compartmentalization.

Decouple We call commands that modify the state of the state machine **writes** and commands that don't modify the state of the state machine **reads**. The leader must process every write because it has to linearize the writes with respect to one another, and *every* replica must process every write because otherwise the replicas' state would diverge (imagine if one replica performs a write but the other replicas don't). However, because reads do not modify the state of the state machine, the leader does not have to linearize them (reads commute), and only a single replica (as opposed to every replica) needs to execute a read.

We take advantage of this observation by decoupling the read path from the write path. Writes are processed as before, but we bypass the leader and perform a read on a single replica by using the ideas from Paxos Quorum Reads (PQR) [13]. Specifically, to perform a read, a client sends a PreRead$\langle\rangle$ message to a read quorum of acceptors. Upon receiving a PreRead$\langle\rangle$ message, an acceptor a_i returns a PreReadAck$\langle\rangle$ message where w_i is the index of the largest log entry in which the acceptor has voted (i.e. the largest log entry in which the acceptor has sent a Phase2B message). We call this w_i a vote watermark. When the client receives PreReadAck messages from a read quorum of acceptors, it computes j as the maximum of all received vote watermarks. It then sends a Read$\langle x, j\rangle$ request to any one of the replicas where x is an arbitrary read (i.e. a command that does not modify the state of the state machine).

When a replica receives a Read$\langle x, j\rangle$ request from a client, it waits until it has executed the command in log entry j. Recall that replicas execute commands in log order, so if the replica has executed the command in log entry j, then it has also executed all of the commands in log entries less than j. After the replica has executed the command in log entry j, it executes x and returns the result to the client. Note that upon receiving a Read$\langle x, j\rangle$ message, a replica may have already executed the log beyond j. That is, it may have already executed the commands in log entries $j + 1$, $j + 2$, and so on. This is okay because as long as the replica has executed the command in log entry j, it is safe to execute x.

Scale The decoupled read and write paths are shown in Figure 3.4. Reads are sent to a row (read quorum) of acceptors, so we can increase the number of rows to decrease the read load on every individual acceptor, eliminating the acceptors as a read bottleneck. Reads are

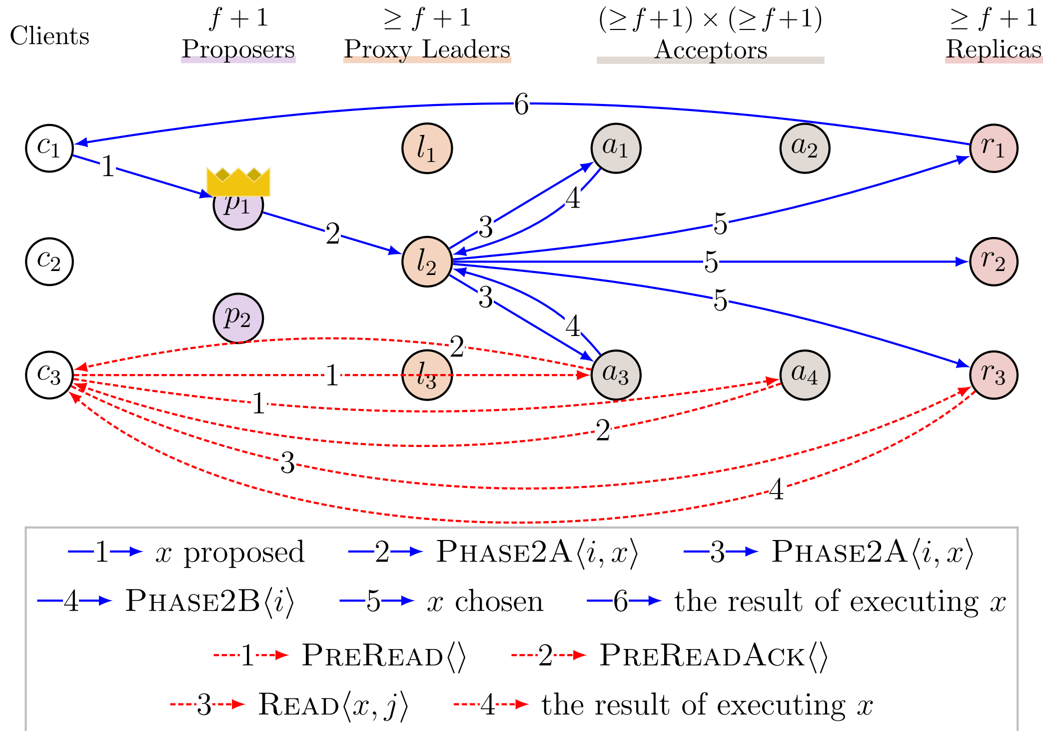


Figure 3.4: An example execution of Compartmentalized MultiPaxos' read and write path ($f = 1$) with a 2×2 acceptor grid. The write path is shown using solid blue lines. The read path is shown using red dashed lines.

also sent to a single replica, so we can increase the number of replicas to eliminate them as a read bottleneck as well.

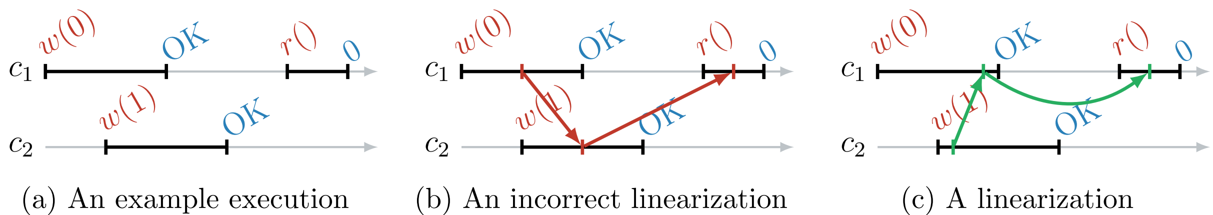
Discussion Note that read-heavy workloads are not a special case. Many workloads are read-heavy [25, 66, 7, 62]. Chubby [11] observes that fewer than 1% of operations are writes, and Spanner [18] observes that fewer than 0.3% of operations are writes.

Also note that increasing the number of columns in an acceptor grid reduces the write load on the acceptors, and increasing the number of rows in an acceptor grid reduces the read load on the acceptors. There is no throughput trade-off between the two. The number of rows and columns can be adjusted independently. Increasing read throughput (by increasing the number of rows) does not decrease write throughput, and vice versa. However, increasing the number of rows does increase the *size* (but not number) of columns, so increasing the number of rows might increase the tail latency of writes, and vice versa.

Correctness We now define linearizability and prove that our protocol implements linearizable reads.

Linearizability is a correctness condition for distributed systems [28]. Intuitively, a linearizable distributed system is indistinguishable from a system running on a single machine that services all requests serially. This makes a linearizable system easy to reason about. We first explain the intuition behind linearizability and then formalize the intuition.

Consider a distributed system that implements a single register. Clients can send requests to the distributed system to read or write the register. After a client sends a read or write request, it waits to receive a response before sending another request. As a result, a client can have at most one operation pending at any point in time.



(a) An example execution (b) An incorrect linearization (c) A linearization

Figure 3.5

As a simple example, consider the execution illustrated in Figure 3.5a where the x-axis represents the passage of time (real time, not logical time [46]). This execution involves two clients, c_1 and c_2. Client c_1 sends a $w(0)$ request to the system, requesting that the value 0 be written to the register. Then, client c_2 sends a $w(1)$ request, requesting that the value 1 be written to the register. The system then sends acknowledgments to c_1 and c_2 before c_1 sends a read request and receives the value 0.

For every client request, let's associate the request with a point in time that falls between when the client sent the request and when the client received the corresponding response. Next, let us imagine that the system executes every request instantaneously at the point in time associated with the request. This hypothetical execution may or may not be consistent with the real execution.

For example, in Figure 3.5a, we have associated every request with a point halfway between its invocation and response. Thus, in this hypothetical execution, the system executes c_1's $w(0)$ request, then c_2's $w(1)$ request, and finally c_1's $r()$ request. In other words, it writes 0 into the register, then 1, and then reads the value 1 (the latest value written). This hypothetical execution is *not* consistent with the real execution because c_1 reads 1 instead of 0.

Now consider the hypothetical execution in Figure 3.5c in which we execute $w(1)$, then $w(0)$, and then $r()$. This execution is consistent with the real execution. Note that c_1 reads 0 in both executions. Such a hypothetical execution—one that is consistent with the real execution—is called a **linearization**. Note that from the clients' perspective, the real execution is indistinguishable from its linearization. Maybe the distributed register really is executing our requests at exactly the points in time that we selected? There's no way for the clients to prove otherwise.

If an execution has a linearization, we say the execution is **linearizable**. Similarly, if a system only allows linearizable executions, we say the system is linearizable. Note that not every execution is linearizable. The execution in Figure 3.6, for example, is not linearizable. Try to find a linearization. You'll see that it's impossible.

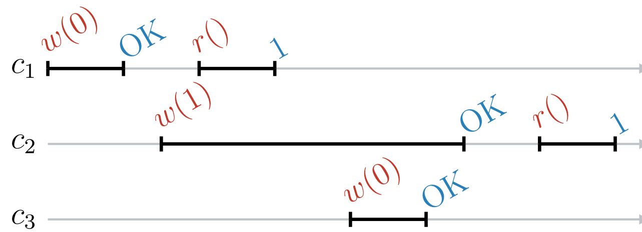


Figure 3.6: An execution that is not linearizable

We now formalize our intuition on linearizability [28]. A **history** is a finite sequence of operation **invocation** and **response** events. For example, the following history:

$$H_{wwr} = c_1.w(0);\ c_2.w(1);\ c_1.\text{OK};\ c_2.\text{OK};\ c_1.r();\ c_1.0$$

is the history illustrated in Figure 3.5a. We draw invocation events in red, and response events in blue. We call an invocation and matching response an **operation**. In H_{wwr}, every invocation is followed eventually by a corresponding response, but this is not always the case. An invocation in a history is **pending** if there does not exist a corresponding response. For example, in the history H_{pending} below, c_2's invocation is pending:

$$H_{\text{pending}} = c_1.w(0);\ c_2.w(1);\ c_1.\text{OK};\ c_1.r();\ c_1.0$$

H_{pending} is illustrated in Figure 3.7. complete(H) is the subhistory of H that only includes non-pending operations. For example,

$$\text{complete}(H_{\text{pending}}) = c_1.w(0);\ c_1.\text{OK};\ c_1.r();\ c_1.0$$

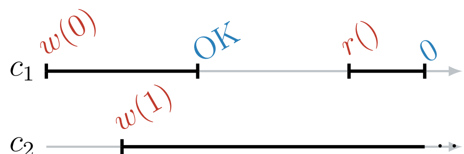


Figure 3.7: A history, H_{pending}, with a pending invocation

A **client subhistory**, $H \mid c_i$, of a history H is the subsequence of all events in H associated with client c_i. Referring again to H_{wwr} above, we have:

$$H_{wwr} \mid c_1 = c_1.w(0);\ c_1.\text{OK};\ c_1.r();\ c_1.0$$
$$H_{wwr} \mid c_2 = c_2.w(1);\ c_2.\text{OK}$$

$H_{wwr} \mid c_1$ is illustrated in Figure 3.8.

Figure 3.8: $H_{wwr} \mid c_1$

Two histories H and H' are **equivalent** if for every client c_i, $H \mid c_i = H' \mid c_i$. For example, consider the following history:

$$H_{wrw} = c_1.w(0);\ c_1.\text{OK};\ c_1.r();\ c_2.w(1);\ c_1.0;\ c_2.\text{OK}$$

H_{wrw} is illustrated in Figure 3.9. H_{wwr} is equivalent to H_{wrw} because

$$H_{wwr} \mid c_1 = c_1.w(0);\ c_1.\text{OK};\ c_1.r();\ c_1.0 = H_{wrw} \mid c_1$$
$$H_{wwr} \mid c_2 = c_2.w(1);\ c_2.\text{OK};\ = H_{wrw} \mid c_2$$

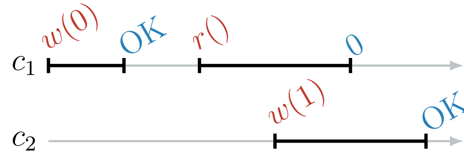


Figure 3.9: H_{wrw}

A history H induces an irreflexive partial order $<_H$ on operations where $o_1 <_H o_2$ if the response of o_1 precedes the invocation of o_2 in H. If $o_1 <_H o_2$, we say o_1 **happens before** o_2. In H_{wwr} for example, c_2's operation happens before c_1's second operation. In H_{wrw}, on the other hand, the two operations are not ordered by the happens before relation. This shows that equivalent histories may not have the same happens before relation.

Finally, a history H is **linearizable** if it can be extended (by appending zero or more response events) to some history H' such that (a) complete(H') is equivalent to some sequential history S, and (b) $<_S$ respects $<_H$ (i.e. if two operations are ordered in H, they must also be ordered in S). S is called a **linearization**. The history H_{wwr}, for example, is linearizable with the linearization

$$S_{wwr} = c_2.w(1);\ c_2.\text{OK};\ c_1.w(0);\ c_1.\text{OK};\ c_1.r();\ c_1.0$$

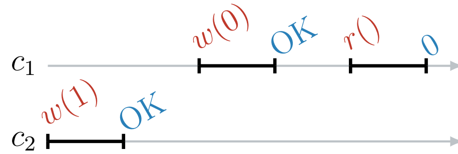


Figure 3.10: S_{wrw}

illustrated in Figure 3.10

We now prove that our protocol correctly implements linearizable reads.

Proof. Let H be an arbitrary history permitted by our protocol. To prove that our protocol is linearizable, we must extend H to a history H' such that complete(H') is equivalent to a sequential history that respects $<_H$.

Recall that extending H to H' is sometimes necessary because of situations like the one shown in Figure 3.11. This example involves a single register with an initial value of 0. c_1 issues a request to write the value of 1, but has not yet received a response. c_2 issues a read request and receives the value 1. If we do not extend the history to include a response to c_1's write, then there will not exist an equivalent sequential history.

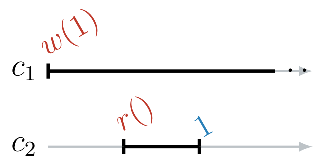


Figure 3.11: A motivating example of history extension

So, which operations should we include in H'? Let k be the largest log index written in or read from in complete(H). First note that for every index $0 \leq i \leq k$, there exists a (potentially pending) write in H that has been chosen in index i. Why? Well, our protocol executes commands in log order, so a write at index k can only complete after all writes with smaller indices have been chosen (and executed by some replica). Similarly, if a read operation reads from slot k, then the write in slot k must have been executed, so again all writes with smaller indices have also been chosen. We extend H to history H' by including responses for all pending write invocations with indices $0 \leq i \leq k$. The responses are formed by executing the $k + 1$ commands in log order.

For example, consider the history G shown in Figure 3.12. w_i represents a write chosen in log index i, r_i represents a read operation that reads from slot i, $w_?$ represents a pending write which has not been chosen in any particular log index, and $r_?$ represents a pending read. complete(G) includes w_1 and r_2, so here $k = 2$ and we must include all writes in indices

0, 1, and 2. That is, we extend G to complete w_0 and w_2. w_4 is left pending, as is $w_?$ and $r_?$. Also note that we could not complete w_4 even if we wanted to because there is no w_3.

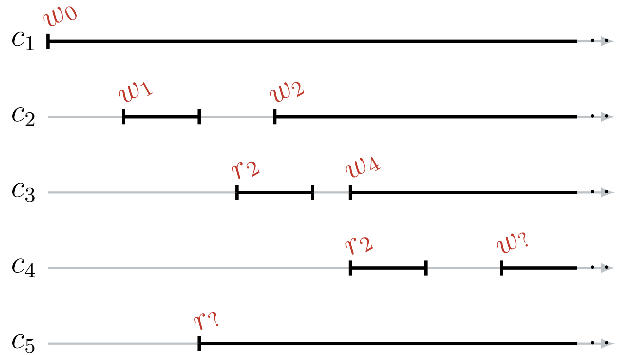


Figure 3.12: An example history G. Responses are not shown, as they are not important for this example.

Now, we must prove that (1) complete(H') is equivalent to some legal sequential history S, and (2) $<_S$ respects $<_H$. We let S be the sequential history formed from executing all writes in log order and from executing every read from index i after the write in index i. If there are multiple reads from index i, the reads are ordered in an arbitrary way that respects $<_H$. For example, the history G in Figure 3.12 has the sequential history S_G shown in Figure 3.13. Note that c_4's read comes after c_3's read. This is essential because we must respect $<_G$. If the two reads were concurrent in G, they could be ordered arbitrarily in S_G.

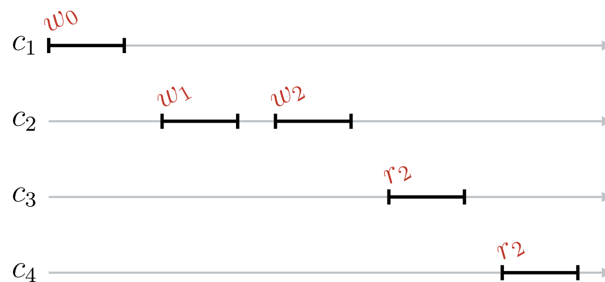


Figure 3.13: A linearization S_G of the history in G Figure 3.12

To prove (1) and (2), we show that if two distinct operations x and y that write to (or read from) log indices i and j are related in H—i.e. $x <_H y$, or x finishes before y begins—then $i \leq j$. We perform a case analysis on whether x and y are reads or writes.

- **x and y are both writes:** At the time x completes in index i, all commands in indices less than i have been chosen because our protocol executes commands in log

order. Thus, when y later begins, it cannot be chosen in a log entry less than i, since every log entry implements consensus. Thus, $i < j$.

- x **and** y **are both reads:** When x completes, command i has been chosen. Thus, some write quorum w of acceptors must have voted for the command in log entry i. When y begins, it sends PreRead messages to some read quorum r of acceptors. r and w intersect, so the client executing y will receive a PreReadAck$\langle w_i \rangle$ message from some acceptor in r with $w_i \geq i$. Therefore, y is guaranteed to read from some $j \geq i$.

- x **is a read and** y **is a write:** When x completes, all commands in indices i and smaller have been chosen. By the first case above, y must be chosen in some index $j > i$.

- x **is a write and** y **is a read:** When x completes, command i has been chosen. As with the second case above, when y begins it will contact an acceptor group with a vote watermark at least as large as i and will subsequently read from at least i.

From this, (1) is immediate since every client's operations are in the same order in H' and in S. (2) holds because S is ordered by log index with ties broken respecting $<_H$, so if $x <_H y$, then $i \leq j$ and $x <_S y$. □

Non-Linearizable Reads Our protocol implements linearizable reads, the strongest form of non-transactional consistency. However, we can extend the protocol to support reads with better performance but weaker consistency. Notably, we can implement sequentially consistent [43] and eventually consistent reads. Writes are always linearizable. The decision of which consistency level to choose depends on the application.

Sequential consistency is a lot like linearizability but without the real-time ordering requirements. Specifically, a history H is sequentially consistent if we can extend it to some history H' such that complete(H') is equivalent to some sequential history S. Unlike with linearizability, we do not require that $<_S$ respects $<_H$.

To implement sequentially consistent reads, every client needs to (a) keep track of the largest log entry it has ever written to or read from, and (b) make sure that all future operations write to or read from a log entry as least as large. Concretely, we make the following changes:

- Every client c_i maintains an integer-valued watermark w_i, initially -1.

- When a replica executes a write w in log entry j and returns the result of executing w to a client c_i, it also includes j. When c_i receives a write index j from a replica, it updates w_i to the max of w_i and j.

- To execute a sequentially consistent read r, a client c_i sends a Read$\langle r, w_i \rangle$ message to any replica. The replica waits until it has executed the write in log entry w_i and then

executes r. It then replies to the client with the result of executing r and the log entry j from which r reads. Here, $j \geq w_i$. When a client receives a read index j, it updates w_i to the max of w_i and j.

Note that a client can finish a sequentially consistent read after one round-trip of communication (in the best case), whereas a linearizable read requires at least two. Moreover, sequentially consistent reads do not involve the acceptors. This means that we can increase read throughput by scaling up the number of replicas without having to scale up the number of acceptors. Also note that sequentially consistent reads are also causally consistent.

Eventually consistent reads are trivial to implement. To execute an eventually consistent read, a client simply sends the read request r directly to any replica. The replica executes the read immediately and returns the result back to the client. Eventually consistent reads do not require any watermark bookkeeping, do not involve acceptors, and never wait for writes. Moreover, the reads are always executed against a consistent prefix of the log.

3.3 Batching

All state machine replication protocols, including MultiPaxos, can take advantage of batching to increase throughput. The standard way to implement batching [77, 76] is to have clients send their commands to the leader and to have the leader group the commands together into batches, as shown in Figure 3.14. The rest of the protocol remains unchanged, with command batches replacing commands. The one notable difference is that replicas now execute one batch of commands at a time, rather than one command at a time. After executing a single command, a replica has to send back a single result to a client, but after executing a batch of commands, a replica has to send a result to every client with a command in the batch.

Compartmentalization 5: Batchers

Bottleneck: *leader*
Decouple: *batch formation and batch sequencing*
Scale: *the number of batchers*

Bottleneck We first discuss write batching and discuss read batching momentarily. Batching increases throughput by amortizing the communication and computation cost of processing a command. Take the acceptors for example. Without batching, an acceptor processes two messages *per command*. With batching, however, an acceptor only processes two messages *per batch*. The acceptors process fewer messages per command as the batch size increases. With batches of size 10, for example, an acceptor processes 10× fewer messages per command with batching than without.

Refer again to Figure 3.14. The load on the proxy leaders and the acceptors both decrease as the batch size increases, but this is not the case for the leader or the replicas. We focus first

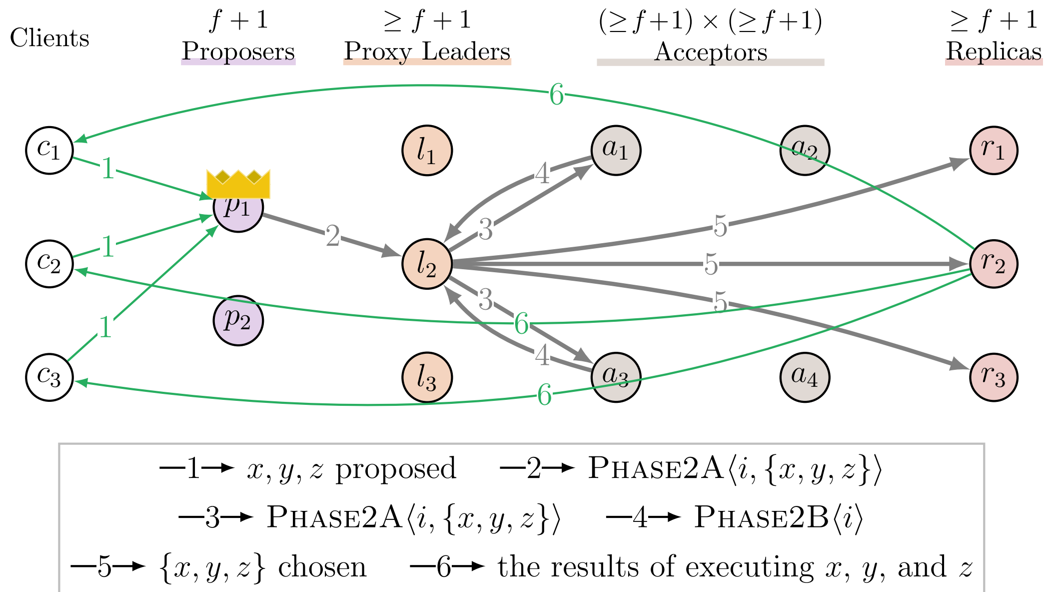


Figure 3.14: An example execution of Compartmentalized MultiPaxos with batching ($f = 1$). Messages that contain a batch of commands, rather than a single command, are drawn thicker. Note how replica r_2 has to send multiple messages after executing a batch of commands.

on the leader. To process a single batch of n commands, the leader has to receive n messages and send one message. Unlike the proxy leaders and acceptors, the leader's communication cost is linear in the number of commands rather than the number of batches. This makes the leader a very likely throughput bottleneck.

Decouple The leader has two responsibilities. It forms batches, and it sequences batches. We decouple the two responsibilities by introducing a set of at least $f + 1$ **batchers**, as illustrated in Figure 3.15. The batchers are responsible for forming batches, while the leader is responsible for sequencing batches.

More concretely, when a client wants to propose a state machine command, it sends the command to a randomly selected batcher (1). After receiving sufficiently many commands from the clients (or after a timeout expires), a batcher places the commands in a batch and forwards it to the leader (2). When the leader receives a batch of commands, it assigns it a log entry, forms a Phase 2a message, and sends the Phase2a message to a proxy leader (3). The rest of the protocol remains unchanged.

Without batchers, the leader has to receive n messages per batch of n commands. With batchers, the leader only has to receive one. This either reduces the load on the bottleneck leader or eliminates it as a bottleneck completely.

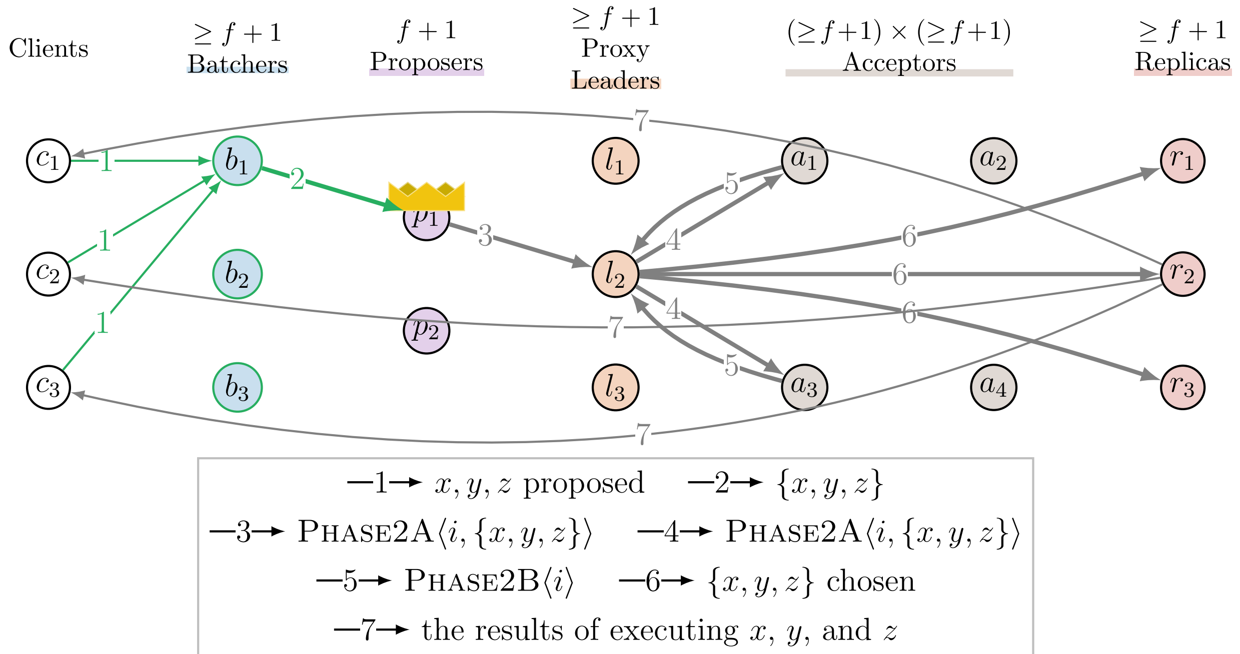


Figure 3.15: An example execution of Compartmentalized MultiPaxos with batchers ($f = 1$).

Scale The batchers are embarrassingly parallel. We can increase the number of batchers until they're not a throughput bottleneck.

Discussion Read batching is very similar to write batching. Clients send reads to randomly selected batchers, and batchers group reads together into batches. After a batcher has formed a read batch X, it sends a PreRead$\langle\rangle$ message to a read quorum of acceptors, computes the resulting watermark i, and sends a Read$\langle X, i\rangle$ request to any one of the replicas.

Compartmentalization 6: Unbatchers

Bottleneck: *replicas*
Decouple: *batch processing and batch replying*
Scale: *the number of unbatchers*

Bottleneck After executing a batch of n commands, a replica has to send n messages back to the n clients. Thus, the replicas (like the leader without batchers) suffer communication overheads linear in the number of commands rather than the number of batches.

Decouple The replicas have two responsibilities. They execute batches of commands, and they send replies to the clients. We decouple these two responsibilities by introducing a set of at least $f + 1$ **unbatchers**, as illustrated in Figure 3.16. The replicas are responsible for executing batches of commands, while the unbatchers are responsible for sending the results of executing the commands back to the clients. Concretely, after executing a batch of commands, a replica forms a batch of results and sends the batch to a randomly selected unbatcher (7). Upon receiving a result batch, an unbatcher sends the results back to the clients (8). This decoupling reduces the load on the replicas.

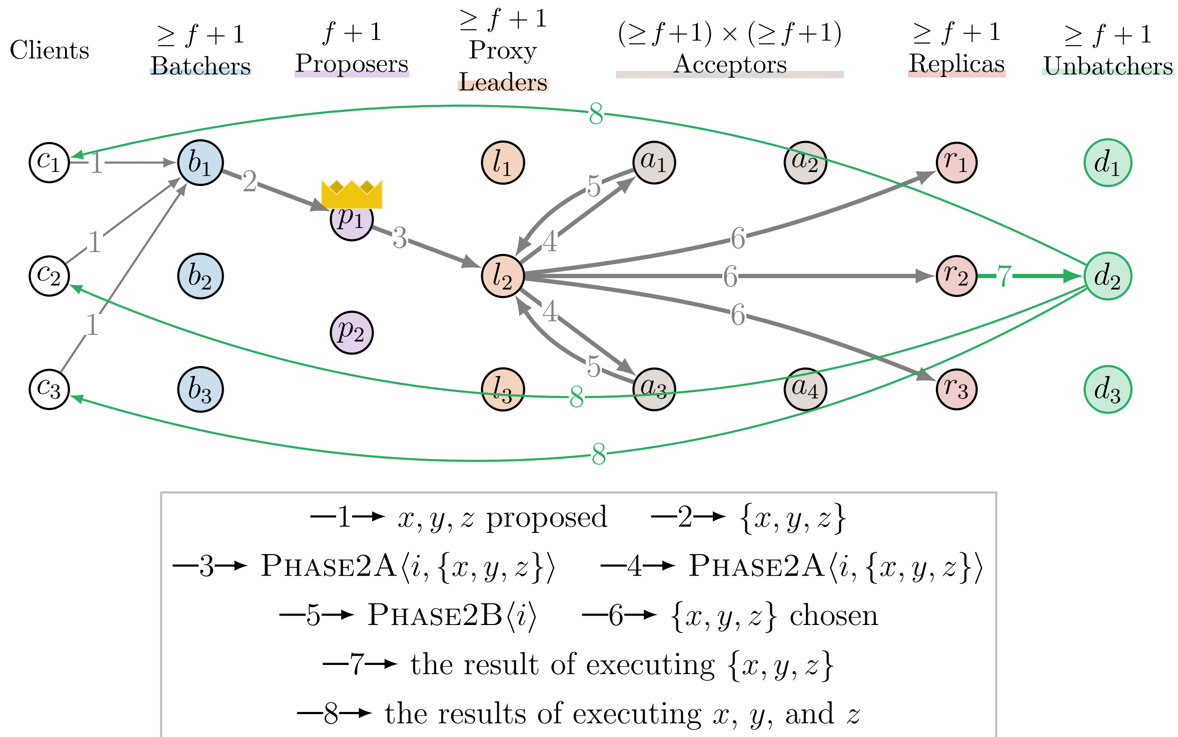


Figure 3.16: An example execution of Compartmentalized MultiPaxos with unbatchers ($f = 1$).

Scale As with batchers, unbatchers are embarrassingly parallel, so we can increase the number of unbatchers until they are not a throughput bottleneck.

Discussion Read unbatching is identical to write unbatching. After executing a batch of reads, a replica forms the corresponding batch of results and sends it to a randomly selected unbatcher.

3.4 Mencius

Background

As discussed previously, the MultiPaxos leader is a throughput bottleneck because all commands go through the leader and because the leader performs disproportionately more work per command than the acceptors or replicas. Mencius is a MultiPaxos variant that attempts to eliminate this bottleneck by using more than one leader.

Rather than having a single leader sequence all commands in the log, Mencius round-robin partitions the log among multiple leaders. For example, consider the scenario with three leaders l_1, l_2, and l_3 illustrated in Figure 3.17. Leader l_1 gets commands chosen in slots 0, 3, 6, etc.; leader l_2 gets commands chosen in slots 1, 4, 7, etc.; and leader l_3 gets commands chosen in slots 2, 5, 8, etc.

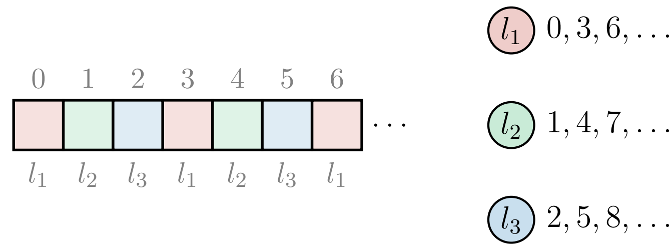


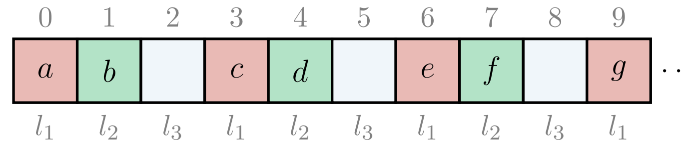
Figure 3.17: A Mencius log round robin partitioned among three leaders.

Having multiple leaders works well when all the leaders process commands at the exact same rate. However, if one of the leaders is slower than the others, then holes start appearing in the log entries owned by the slow leader. This is illustrated in Figure 3.18a. Figure 3.18a depicts a Mencius log partitioned across three leaders. Leaders l_1 and l_2 have both gotten a few commands chosen (e.g., a in slot 0, b in slot 1, etc.), but leader l_3 is lagging behind and has not gotten any commands chosen yet. Replicas execute commands in log order, so they are unable to execute all of the chosen commands until l_3 gets commands chosen in its vacant log entries.

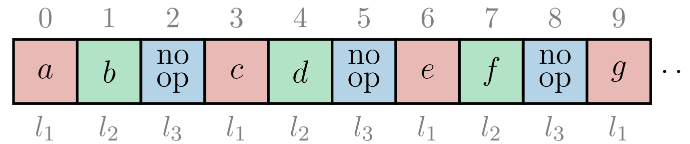
If a leader detects that it is lagging behind, then it fills its vacant log entries with a sequence of noops. A **noop** is a distinguished command that does not affect the state of the replicated state machine. In Figure 3.18b, we see that l_3 fills its vacant log entries with noops. This allows the replicas to execute all of the chosen commands.

More concretely, a Mencius deployment that tolerates f faults is implemented with $2f+1$ **servers**, as illustrated in Figure 3.19. Roughly speaking, every Mencius server plays the role of a MultiPaxos leader, acceptor, and replica.

When a client wants to propose a state machine command x, it sends x to any of the servers (1). Upon receiving command x, a server s_l plays the role of a leader. It assigns the command x a slot i and sends a Phase 2a message that includes x and i to the other servers



(a) Before noops



(b) After noops

Figure 3.18: An example of using noops to deal with a slow leader. Leader l_3 is slower than leaders l_1 and l_2, so the log has holes in l_3's slots. l_3 fills its holes with noops to allow commands in the log to be executed.

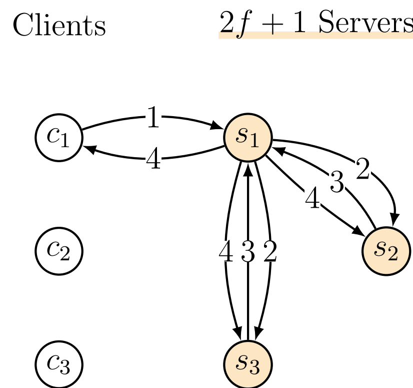


Figure 3.19: An example execution of Mencius.

(2). Upon receiving a Phase 2a message, a server s_a plays the role of an acceptor and replies with a Phase 2b message (3).

In addition, s_a uses i to determine if it is lagging behind s_l. If it is, then it sends a SKIP message along with the Phase 2b message. The SKIP message informs the other servers to choose a noop in every slot owned by s_a up to slot i. For example, if a server s_a's next available slot is slot 10 and it receives a Phase 2a message for slot 100, then it broadcasts a SKIP message informing the other servers to place noops in all of the slots between slots 10 and 100 that are owned by server s_a. Mencius leverages a protocol called Coordinated Paxos to ensure noops are chosen correctly. We refer to the reader to [57] for details.

Upon receiving Phase 2b messages for command x from a majority of the servers, server s_l deems the command x chosen. It informs the other servers that the command has been

chosen and also sends the result of executing x back to the client.

Compartmentalization

Mencius uses multiple leaders to avoid being bottlenecked by a single leader. However, despite this, Mencius still does not achieve optimal throughput. Part of the problem is that every Mencius server plays three roles, that of a leader, an acceptor, and a replica. Because of this, a server has to send and receive a total of roughly $3f + 5$ messages for every command that it leads *and also* has to send and receive messages acking other servers as they simultaneously choose commands.

We can solve this problem by decoupling the servers. Instead of deploying a set of heavily loaded servers, we instead view Mencius as a MultiPaxos variant and deploy it as a set of proposers, a set of acceptors, and set of replicas. This is illustrated in Figure 3.20.

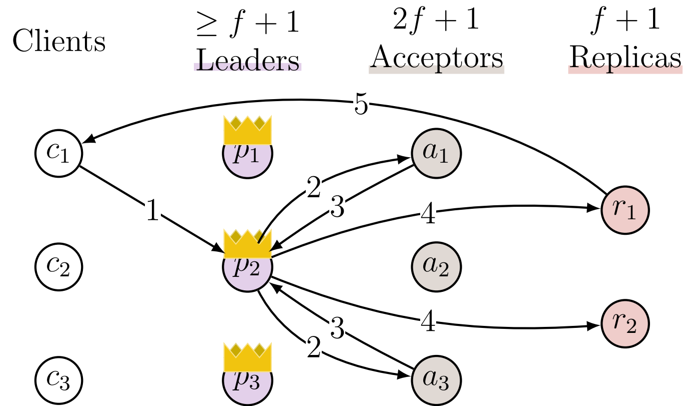


Figure 3.20: An example execution of decoupled Mencius. Note that every proposer is a leader.

Now, Mencius is equivalent to MultiPaxos with the following key differences. First, every proposer is a leader, with the log round-robin partitioned among all the proposers. If a client wants to propose a command, it can send it to any of the proposers. Second, the proposers periodically broadcast their next available slots to one another. Every server uses this information to gauge whether it is lagging behind. If it is, it chooses noops in its vacant slots, as described above.

This decoupled Mencius is a step in the right direction, but it shares many of the problems that MultiPaxos faced. The proposers are responsible for both sequencing commands and for coordinating with acceptors; we have a single unscalable group of acceptors; and we are deploying too few replicas. Thankfully, we can compartmentalize Mencius in exactly the same way as MultiPaxos by leveraging proxy leaders, acceptor grids, and more replicas. This is illustrated in Figure 3.21.

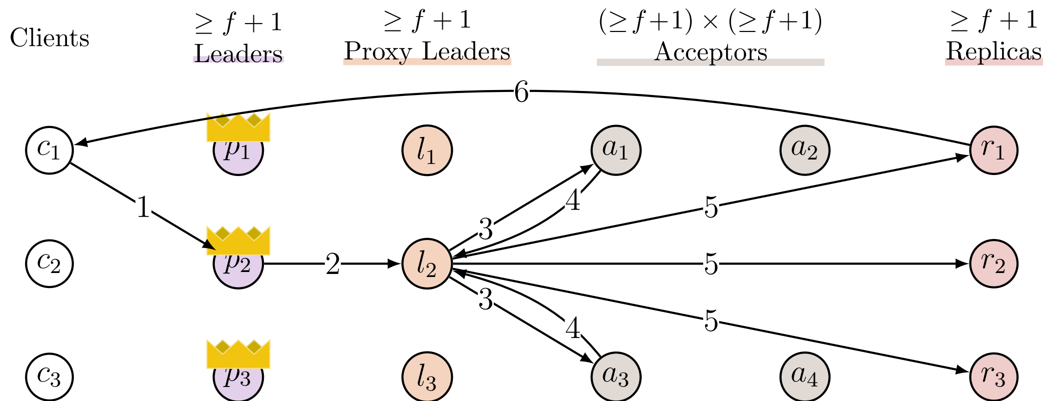


Figure 3.21: An execution of Mencius with proxy leaders, acceptor grids, and an increased number of replicas.

This protocol shares all of the advantages of compartmentalized MultiPaxos. Proxy leaders and acceptors both trivially scale so are not bottlenecks, while leaders and replicas have been pared down to their essential responsibilities of sequencing and executing commands respectively. Moreover, because Mencius allows us to deploy multiple leaders, we can also increase the number of leaders until they are no longer a bottleneck. We can also introduce batchers and unbatchers like we did with MultiPaxos and can implement linearizable leaderless reads.

3.5 S-Paxos

Background

S-Paxos [10] is a MultiPaxos variant that, like Mencius, aims to avoid being bottlenecked by a single leader. Recall that when a MultiPaxos leader receives a state machine command x from a client, it broadcasts a Phase 2a message to the acceptors that includes the command x. If the leader receives a state machine command that is large (in terms of bytes) or receives a large batch of modestly sized commands, the overheads of disseminating the commands begin to dominate the cost of the protocol, exacerbating the fact that command disseminating is performed solely by the leader.

S-Paxos avoids this by decoupling command dissemination from command sequencing—separating control from from data flow—and distributing command dissemination across all nodes. More concretely, an S-Paxos deployment that tolerates f faults consists of $2f + 1$ servers, as illustrated in Figure 3.22. Every server plays the role of a MultiPaxos proposer, acceptor, and replica. It also plays the role of a **disseminator** and **stabilizer**, two roles that will become clear momentarily.

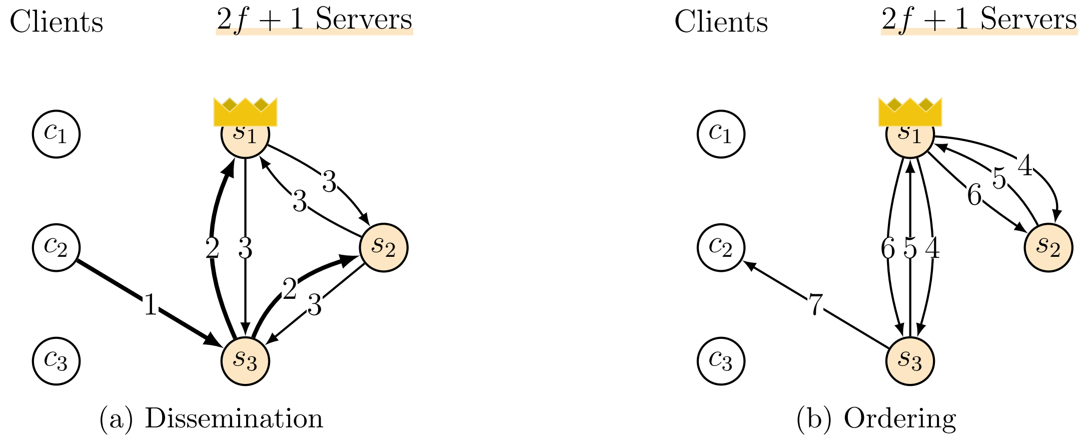


Figure 3.22: An example execution of S-Paxos. Messages that include client commands (as opposed to ids) are bolded.

When a client wants to propose a state machine command x, it sends x to any of the servers. Upon receiving a command from a client, a server plays the part of a disseminator. It assigns the command a globally unique id id_x and begins a **dissemination phase** with the goal of persisting the command and its id on at least a majority of the servers. This is shown in Figure 3.22a. The server broadcasts x and id_x to the other servers. Upon receiving x and id_x, a server plays the role of a stabilizer and stores the pair in memory. It then broadcasts an acknowledgement to all servers. The acknowledgement contains id_x but not x.

One of the servers is the MultiPaxos leader. Upon receiving acknowledgements for id_x from a majority of the servers, the leader knows the command is stable. It then uses the id id_x as a proxy for the corresponding command x and runs the MultiPaxos protocol as usual (i.e. broadcasting Phase 2a messages, receiving Phase 2b messages, and notifying the other servers when a command id has been chosen) as shown in Figure 3.22b. Thus, while MultiPaxos agrees on a log of *commands*, S-Paxos agrees on a log of *command ids*.

The S-Paxos leader, like the MultiPaxos leader, is responsible for ordering command ids and getting them chosen. But, the responsibility of disseminating commands is shared by all the servers.

Compartmentalization

We compartmentalize S-Paxos similar to how we compartmentalize MultiPaxos and Mencius. First, we decouple servers into a set of at least $f+1$ disseminators, a set of $2f+1$ stabilizers, a set of proposers, a set of acceptors, and a set of replicas. This is illustrated in Figure 3.23. To propose a command x, a client sends it to any of the disseminators. Upon receiving x, a disseminator persists the command and its id id_x on at least a majority of (and typically all

of) the stabilizers. It then forwards the id to the leader. The leader gets the id chosen in a particular log entry and informs one of the stabilizers. Upon receiving id_x from the leader, the stabilizer fetches x from the other stabilizers if it has not previously received it. The stabilizer then informs the replicas that x has been chosen. Replicas execute commands in prefix order and reply to clients as usual.

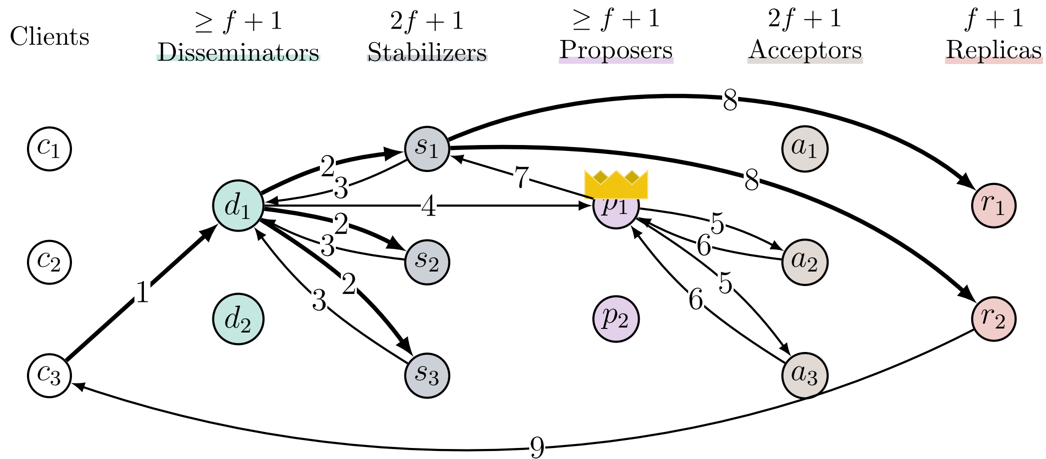


Figure 3.23: An example execution of decoupled S-Paxos. Messages that include client commands (as opposed to ids) are bolded. Note that the MultiPaxos leader does not send or receive any messages that include a command, only messages that include command ids.

Though S-Paxos relieves the MultiPaxos leader of its duty to broadcast commands, the leader still has to broadcast command ids. In other words, the leader is no longer a bottleneck on the data path but is still a bottleneck on the control path. Moreover, disseminators and stabilizers are potential bottlenecks. We can resolve these issues by compartmentalizing S-Paxos similar to how we compartmentalized MultiPaxos. We introduce proxy leaders, acceptor grids, and more replicas. Moreover, we can trivially scale up the number of disseminators; we can deploy stabilizer grids; and we can implement linearizable leaderless reads. This is illustrated in Figure 3.24. To support batching, we can again introduce batchers and unbatchers.

3.6 Evaluation

Latency-Throughput

Experiment Description We call MultiPaxos with the six compartmentalizations described in this chapter **Compartmentalized MultiPaxos**. We implemented MultiPaxos, Compartmentalized MultiPaxos, and an unreplicated state machine in Scala using the Netty

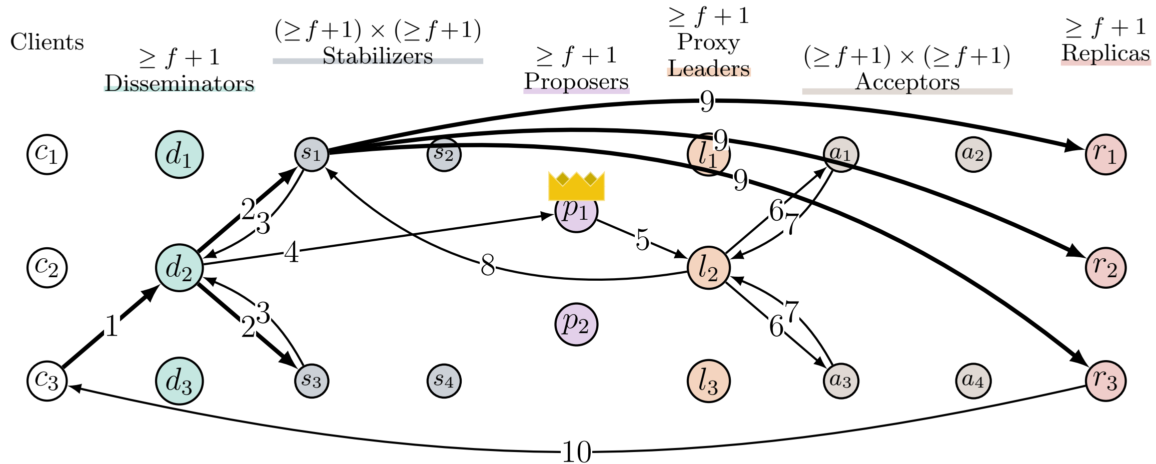
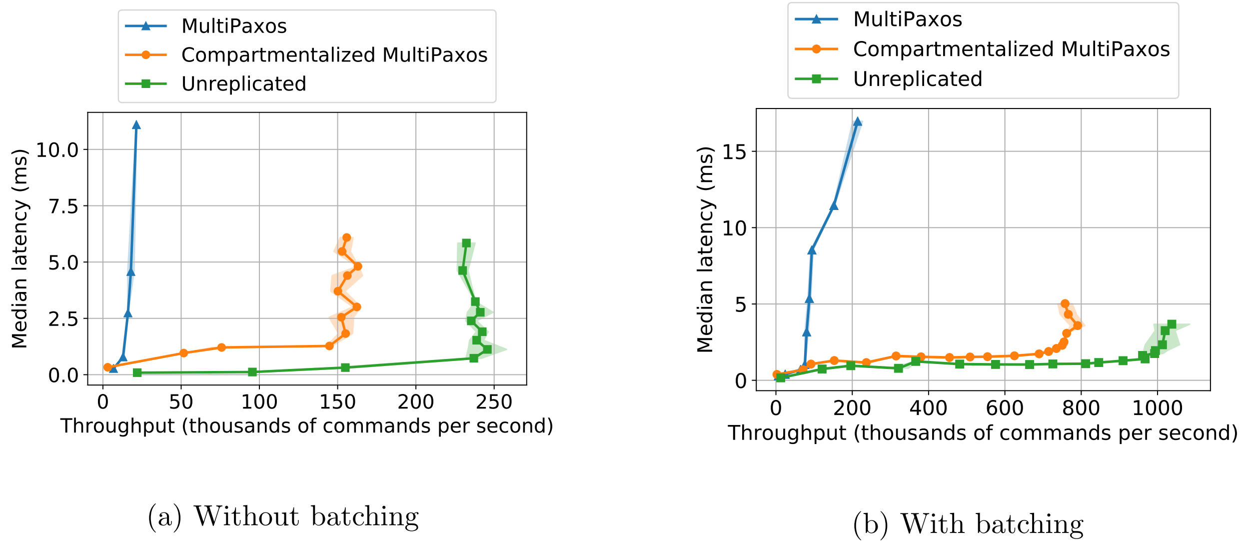


Figure 3.24: An example execution of S-Paxos with stabilizer grids, proxy leaders, acceptor grids, and an increased number of replicas. Messages that include client commands (as opposed to ids) are bolded.

networking library (see github.com/mwhittaker/frankenpaxos). MultiPaxos employs $2f + 1$ machines with each machine playing the role of a MultiPaxos proposer, acceptor, and replica. The unreplicated state machine is implemented as a single process on a single server. Clients send commands directly to the state machine. Upon receiving a command, the state machine executes the command and immediately sends back the result. Note that unlike MultiPaxos and Compartmentalized MultiPaxos, the unreplicated state machine is *not* fault tolerant. If the single server fails, all state is lost and no commands can be executed. Thus, the unreplicated state machine should not be viewed as an apples-to-apples comparison with the other two protocols. Instead, the unreplicated state machine sets an upper bound on attainable performance.

We measure the throughput and median latency of the three protocols under workloads with a varying numbers of clients. Each client issues state machine commands in a closed loop. It waits to receive the result of executing its most recently proposed command before it issues another. All three protocols replicate a key-value store state machine where the keys are integers and the values are 16 byte strings. In this benchmark, all state machine commands are writes. There are no reads. Note that multiple clients are run within a single process, so the number of physical client processes can be significantly less than the number of logical clients.

We deploy the protocols with and without batching for $f = 1$. Without batching, we deploy Compartmentalized MultiPaxos with two proposers, ten proxy leaders, a two by two grid of acceptors, and four replicas. With batching, we deploy two batchers, two proposers, three proxy replicas, a simple majority quorum system of three acceptors, two replicas, and three unbatchers. For simplicity, every node is deployed on its own machine, but in practice,



(a) Without batching

(b) With batching

Figure 3.25: The latency and throughput of MultiPaxos, Compartmentalized MultiPaxos, and an unreplicated state machine.

nodes can be physically co-located. In particular, any two logical roles can be placed on the same machine without violating fault tolerance constraints, so long as the two roles are not the same.

We deploy the three protocols on AWS using a set of m5.xlarge machines within a single availability zone. Every m5.xlarge instance has 4 vCPUs and 16 GiB of memory. Everything is done in memory, and nothing is written to disk (because everything is replicated, data is persistent even without writing it to disk). In our experiments, the network is never a bottleneck. All numbers presented are the average of three executions of the benchmark. As is standard, we implement MultiPaxos and Compartmentalized MultiPaxos with thriftiness enabled [62]. For a given number of clients, the batch size is set empirically to optimize throughput. For a fair comparison, we deploy the unreplicated state machine with a set of batchers and unbatchers when batching is enabled.

Results The results of the experiment are shown in Figure 3.25. The standard deviation of throughput measurements are shown as a shaded region. Without batching, MultiPaxos has a peak throughput of roughly 25,000 commands per second, while Compartmentalized MultiPaxos has a peak throughput of roughly 150,000 commands per second, a $6\times$ increase. The unreplicated state machine outperforms both protocols. It achieves a peak throughput of roughly 250,000 commands per second. Compartmentalized MultiPaxos underperforms the unreplicated state machine because—despite decoupling the leader as much as possible—the single leader remains a throughput bottleneck. Note that after fully compartmentalizing MultiPaxos, either the leader or the replicas are guaranteed to be the throughput bottle-

neck because all other components (e.g., proxy leaders, acceptors, batchers, unbatchers) can be scaled arbitrarily. Implementation and deployment details (e.g., what state machine is being replicated) determine which component is the ultimate throughput bottleneck. All three protocols have millisecond latencies at peak throughput. With batching, MultiPaxos, Compartmentalized MultiPaxos, and the unreplicated state machine have peak throughputs of roughly 200,000, 800,000 and 1,000,000 commands per second respectively.

Compartmentalized MultiPaxos uses 6.66× more machines than MultiPaxos. On the surface, this seems like a weakness, but in reality it is a strength. MultiPaxos does not scale, so it is unable to take advantage of more machines. Compartmentalized MultiPaxos, on the other hand, achieves a 6× increase in throughput using 6.66× the number of resources. Thus, we achieve 90% of perfect linear scalability. In fact, with the mixed read-write workloads below, we are able to scale throughput superlinearly with the number of resources. This is because compartmentalization eliminates throughput bottlenecks. With throughput bottlenecks, non-bottlenecked components are underutilized. When we eliminate the bottlenecks, we eliminate underutilization and can increase performance without increasing the number of resources. Moreover, a protocol does not have to be *fully* compartmentalized. We can selectively compartmentalize some but not all throughput bottlenecks to reduce the number of resources needed. In other words, MultiPaxos and Compartmentalized MultiPaxos are not two alternatives, but rather two extremes in a trade-off between throughput and resource usage.

We also compared the unbatched performance of Compartmentalized MultiPaxos and the unreplicated state machine with values being 100 bytes and 1000 bytes. The results are shown in Figure 3.26. Expectedly, the protocols' peak throughput decreases as we increase the value size.

Ablation Study

Experiment Description We now perform an ablation study to measure the effect of each compartmentalization. In particular, we begin with MultiPaxos and then decouple and scale the protocol according to the six compartmentalizations, measuring peak throughput along the way. Note that we cannot measure the effect of each individual compartmentalization in isolation because decoupling and scaling a component only improves performance if that component is a bottleneck. Thus, to measure the effect of each compartmentalization, we have to apply them all, and we have to apply them in an order that is consistent with the order in which bottlenecks appear. All the details of this experiment are the same as the previous experiment unless otherwise noted.

Results The unbatched ablation study results are shown in Figure 3.27a. MultiPaxos has a throughput of roughly 25,000 commands per second. When we decouple the protocol and introduce proxy leaders (Section 3.2), we increase the throughput to roughly 70,000 commands per second. This decoupled MultiPaxos uses the bare minimum number of proposers (2), proxy leaders (2), acceptors (3), and replicas (2). We then scale up the number of proxy

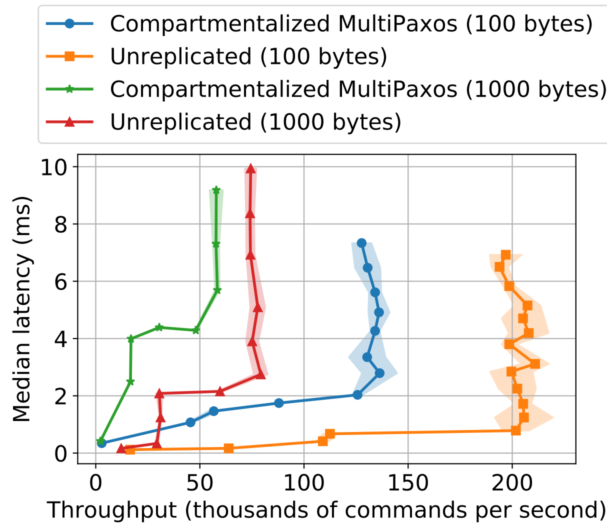
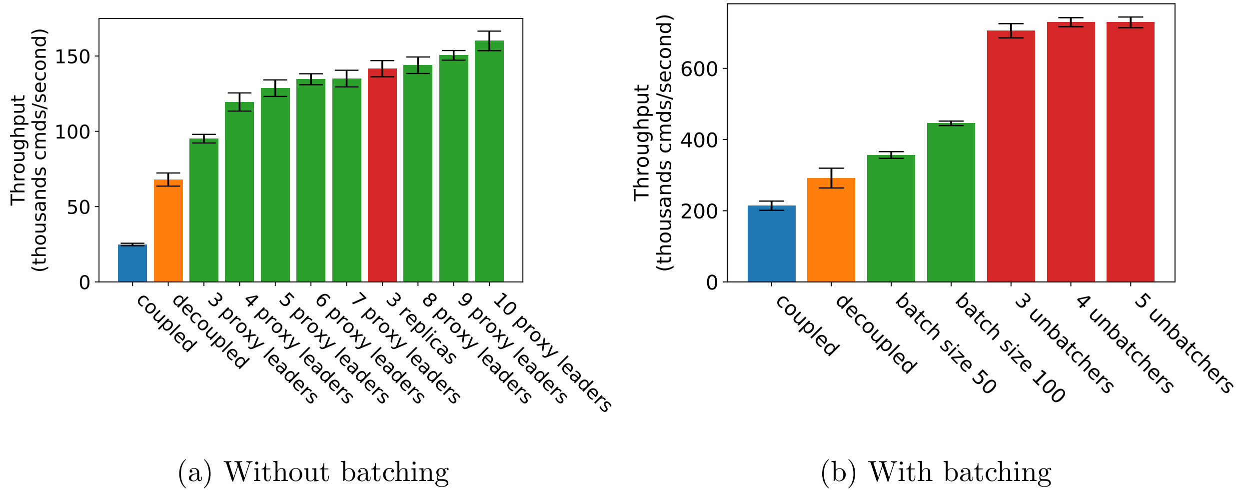


Figure 3.26: The latency and throughput of Compartmentalized MultiPaxos and an unreplicated state machine without batching and with larger value sizes.



(a) Without batching

(b) With batching

Figure 3.27: An ablation study. Standard deviations are shown using error bars.

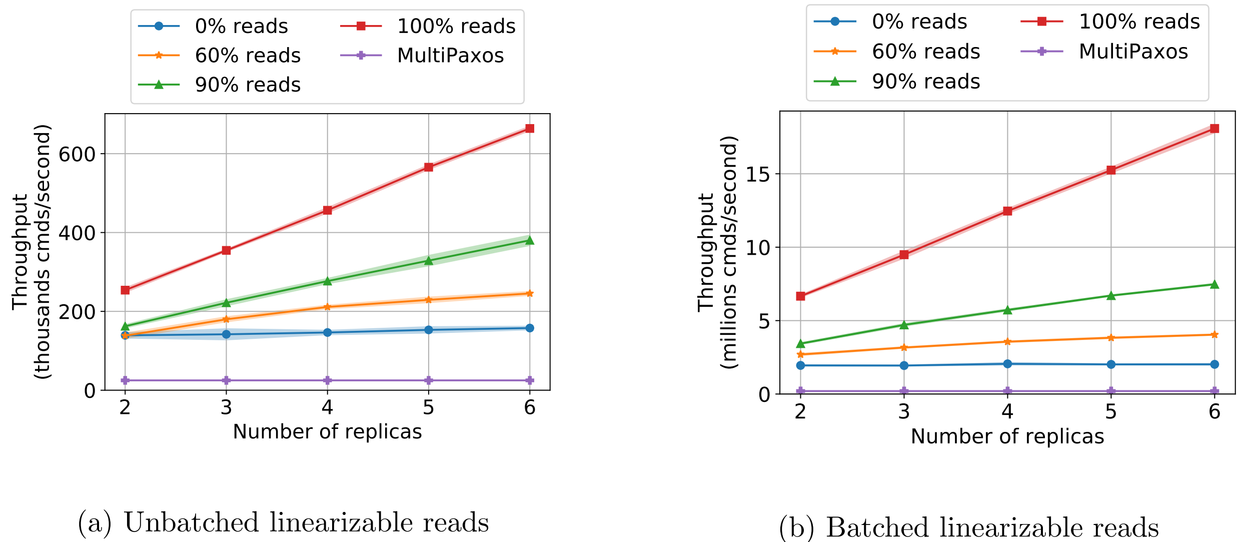
leaders from 2 to 7. The proxy leaders are the throughput bottleneck, so as we scale them up, the throughput of the protocol increases until it plateaus at roughly 135,000 commands per second. At this point, the proxy leaders are no longer the throughput bottleneck; the replicas are. We introduce an additional replica (Section 3.2), though the throughput does not increase. This is because proxy leaders broadcast commands to all replicas, so introducing a new replica increases the load on the proxy leaders making them the bottleneck again.

We then increase the number of proxy leaders to 10 to increase the throughput to roughly 150,000 commands per second. At this point, we determined empirically that the leader was the bottleneck. In this experiment, the acceptors are never the throughput bottleneck, so increasing the number of acceptors does not increase the throughput (Section 3.2). However, this is particular to our write-only workload. In the mixed read-write workloads discussed momentarily, scaling up the number of acceptors is critical for high throughput.

The batched ablation study results are shown in Figure 3.27b. We decouple MultiPaxos and introduce two batchers and two unbatchers with a batch size of 10 (Section 3.3, Section 3.3). This increases the throughput of the protocol from 200,000 commands per second to 300,000 commands per second. We then increase the batch size to 50 and then to 100. This increases throughput to 500,000 commands per second. We then increase the number of unbatchers to 3 and reach a peak throughput of roughly 800,000 commands per second. For this experiment, two batchers and three unbatchers are sufficient to handle the clients' load. With more clients and a larger load, more batchers would be needed to maximize throughput.

Compartmentalization allows us to decouple and scale protocol components, but it doesn't automatically tell us the extent to which we should decouple and scale. Understanding this, through ablation studies like the one presented here, must currently be done by hand. As a line of future work, we are researching how to automatically deduce the optimal amount of decoupling and scaling.

Read Scalability



(a) Unbatched linearizable reads

(b) Batched linearizable reads

Figure 3.28: Peak throughput vs the number of replicas

Experiment Description Thus far, we have looked at write-only workloads. We now measure the throughput of Compartmentalized MultiPaxos under a workload with reads *and* writes. In particular, we measure how the throughput of Compartmentalized MultiPaxos scales as we increase the number of replicas. We deploy Compartmentalized MultiPaxos with and without batching; with 2, 3, 4, 5, and 6 replicas; and with workloads that have 0%, 60%, 90%, and 100% reads. For any given workload and number of replicas, proxy leaders, and acceptors is chosen to maximize throughput. The batch size is 50. In the batched experiments, we do *not* use batchers and unbatchers. Instead, clients form batches of commands themselves. This has no effect on the throughput measurements. We did this only to reduce the number of client machines that we needed to saturate the system. This was not an issue with the write-only workloads because they had significantly lower peak throughputs.

Results The unbatched results are shown in Figure 3.28a. We also show MultiPaxos' throughput for comparison. MultiPaxos does not distinguish reads and writes, so there is only a single line to compare against. With a 0% read workload, Compartmentalized MultiPaxos has a throughput of roughly 150,000 commands per second, and the protocol does not scale much with the number of replicas. This is consistent with our previous experiments. For workloads with reads and writes, our results confirm two expected trends. First, the higher the fraction of reads, the higher the throughput. Second, the higher the fraction of reads, the better the protocol scales with the number of replicas. With a 100% read workload, for example, Compartmentalized MultiPaxos scales linearly up to a throughput of roughly 650,000 commands per second with 6 replicas. The batched results, shown in Figure 3.28b, are very similar. With a 100% read workload, Compartmentalized MultiPaxos scales linearly up to a throughput of roughly 17.5 million commands per second.

Our results also show two *counterintuitive* trends. First, a small increase in the fraction of writes can lead to a disproportionately large decrease in throughput. For example, the throughput of the 90% read workload is far less than 90% of the throughput of the 100% read workload. Second, besides the 100% read workload, throughput does *not* scale linearly with the number of replicas. We see that the throughput of the 0%, 60%, and 90% read workloads scale sublinearly with the number of replicas. These results are not an artifact of our protocol; they are fundamental. Any state machine replication protocol where writes are processed by every replica and where reads are processed by a single replica [85, 98, 13] will exhibit these same two performance anomalies.

We can explain this analytically. Assume that we have n replicas; that every replica can process at most α commands per second; and that we have a workload with a f_w fraction of writes and a $f_r = 1 - f_w$ fraction of reads. Let T be peak throughput, measured in commands per second. Then, our protocol has a peak throughput of $f_w T$ writes per second and $f_r T$ reads per second. Writes are processed by *every* replica, so we impose a load of $n f_w T$ writes per second on the replicas. Reads are processed by *a single* replica, so we impose a load of $f_r T$ reads per second on the replicas. The total aggregate throughput of the system is $n\alpha$,

so we have $n\alpha = nf_wT + f_rT$. Solving for T, we find the peak throughput of our system is

$$\frac{n\alpha}{nf_w + f_r}$$

This formula is plotted in Figure 3.29 with $\alpha = 100,000$. The limit of our peak throughput as n approaches infinity is $\frac{\alpha}{f_w}$. This explains both of the performance anomalies described above. First, it shows that peak throughput has a $\frac{1}{f_w}$ relationship with the fraction of writes, meaning that a small increase in f_w can have a large impact on peak throughput. For example, if we increase our write fraction from 1% to 2%, our throughput will half. A 1% change in write fraction leads to a 50% reduction in throughput. Second, it shows that throughput does not scale linearly with the number of replicas; it is upper bounded by $\frac{\alpha}{f_w}$. For example, a workload with 50% writes can never achieve more than twice the throughput of a 100% write workload, even with an infinite number of replicas.

The results for sequentially consistent and eventually consistent reads are shown in Figure 3.30. The throughput of these weakly consistent reads are similar to that of linearizable reads, but they can be performed with far fewer acceptors.

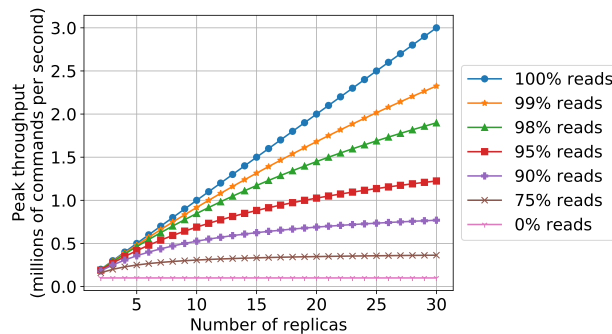
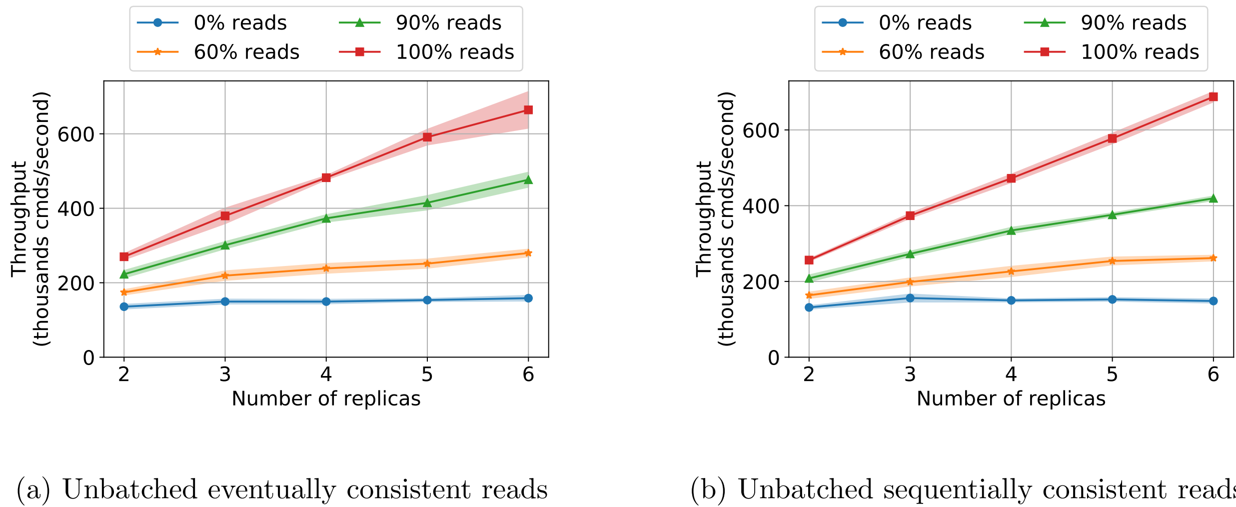


Figure 3.29: Analytical throughput vs the number of replicas.

Skew Tolerance

Experiment Description CRAQ [85] is a chain replication [88] variant with scalable reads. A CRAQ deployment consists of at least $f + 1$ nodes arranged in a linked list, or chain. Writes are sent to the head of the chain and propagated node-by-node down the chain from the head to the tail. When the tail receives the write, it sends a write acknowledgement to its predecessor, and this ack is propagated node-by-node backwards through the chain until it reaches the head. Reads are sent to any node. When a node receives a read of key k, it checks to see if it has any unacknowledged write to that key. If it doesn't, then it performs the read and replies to the client immediately. If it does, then it forwards the read to the tail of the chain. When the tail receives a read, it executes the read immediately and replies to the client.



(a) Unbatched eventually consistent reads

(b) Unbatched sequentially consistent reads

Figure 3.30: Peak throughput vs the number of replicas

We now compare Compartmentalized MultiPaxos with our implementation of CRAQ. In particular, we show that CRAQ (and similar protocols like Harmonia [98]) are sensitive to data skew, whereas Compartmentalized MultiPaxos is not. We deploy Compartmentalized MultiPaxos with two proposers, three proxy leaders, twelve acceptors, and six replicas, and we deploy CRAQ with six chain nodes. Though, our results hold for deployments with a different number of machines as well, as long as the number of Compartmentalized MultiPaxos replicas is equal to the number of CRAQ chain nodes. Both protocols replicate a key-value store with 10,000 keys in the range $1, \ldots, 10,000$. We subject both protocols to the following workload. A client repeatedly flips a weighted coin, and with probability p chooses to read or write to key 1. With probability $1 - p$, it decides to read or write to some other key $2, \ldots, 10,000$ chosen uniformly at random. The client then decides to perform a read with 95% probability and a write with 5% probability. As we vary the value of p, we vary the skew of the workload. When $p = 0$, the workload is completely uniform, and when $p = 1$, the workload consists of reads and writes to a single key. This artificial workload allows to study the effect of skew in a simple way without having to understand more complex skewed distributions.

Results The results are shown in Figure 3.31, with p on the x-axis. The throughput of Compartmentalized MultiPaxos is constant; it is independent of p. This is expected because Compartmentalized MultiPaxos is completely agnostic to the state machine that it is replicating and is completely unaware of the notion of keyed data. Its performance is only affected by the ratio of reads to writes and is completely unaffected by what data is actually being read or written. CRAQ, on the other hand, is susceptible to skew. As we increase skew from $p = 0$ to $p = 1$, the throughput decreases from roughly 300,000 commands per

second to roughly 100,000 commands per second. As we increase p, we increase the fraction of reads which are forwarded to the tail. In the extreme, all reads are forwarded to the tail, and the throughput of the protocol is limited to that of a single node (i.e. the tail).

However, with low skew, CRAQ can perform reads in a single round trip to a single chain node. This allows CRAQ to implement reads with lower latency and with fewer nodes than Compartmentalized MultiPaxos. However, we also note that Compartmentalized MultiPaxos outperforms CRAQ in our benchmark even with no skew. This is because every chain node must process four messages per write, whereas Compartmentalized MultiPaxos replicas only have to process two. CRAQ's write latency also increases with the number of chain nodes, creating a hard trade-off between read throughput and write latency. Ultimately, neither protocol is strictly better than the other. For very read-heavy workloads with low-skew, CRAQ will likely outperform Compartmentalized MultiPaxos using fewer machines, and for workloads with more writes or more skew, Compartmentalized MultiPaxos will likely outperform CRAQ. For the 95% read workload in our experiment, Compartmentalized MultiPaxos has strictly better throughput than CRAQ across all skews, but this is not true for workloads with a higher fraction of reads.

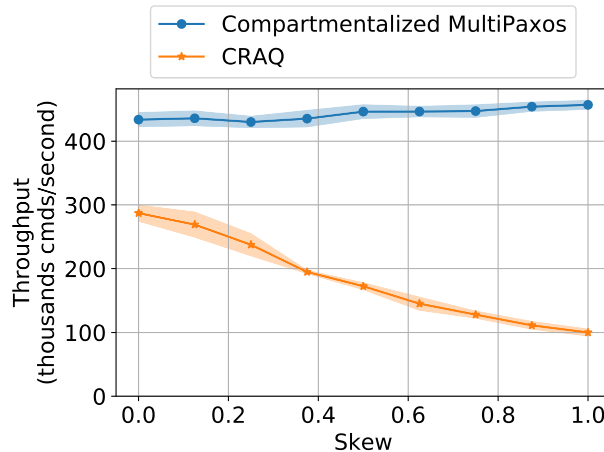


Figure 3.31: The effect of skew on Compartmentalized MultiPaxos and CRAQ.

3.7 Related Work

MultiPaxos Unlike state machine replication protocols like Raft [68] and Viewstamped Replication [53], MultiPaxos [45, 44, 87] is designed with the roles of proposer, acceptor, and replicas logically decoupled. This decoupling alone is not sufficient for MultiPaxos to achieve the best possible throughput, but the decoupling allows for the compartmentalizations described in this chapter.

PigPaxos PigPaxos [14] is a MultiPaxos variant that alters the communication flow between the leader and the acceptors to improve scalability and throughput. Similar to compartmentalization, PigPaxos realizes that the leader is doing many different jobs and is a bottleneck in the system. In particular, PigPaxos substitutes direct leader-to-acceptor communication with a relay network. In PigPaxos the leader sends a message to one or more randomly selected relay nodes, and each relay rebroadcasts the leader's message to the peers in its relay-group and waits for some threshold of responses. Once each relay receives enough responses from its peers, it aggregates them into a single message to reply to the leader. The leader selects a new set of random relays for each new message to prevent faulty relays from having a long-term impact on the communication flow. PigPaxos relays are comparable to our proxy leaders, although the relays are simpler and only alter the communication flow. As such, the relays cannot generally take over the other leader roles, such as quorum counting or replying to the clients. Unlike PigPaxos, whose main goal is to grow to larger clusters, compartmentalization is more general and improves throughput under different conditions and situations.

Chain Replication Chain Replication [88] is a state machine replication protocol in which the set of state machine replicas are arranged in a totally ordered chain. Writes are propagated through the chain from head to tail, and reads are serviced exclusively by the tail. Chain Replication has high throughput compared to MultiPaxos because load is more evenly distributed between the replicas, but every replica must process four messages per command, as opposed to two in Compartmentalized MultiPaxos. The tail is also a throughput bottleneck for read-heavy workloads. Finally, Chain Replication is not tolerant to network partitions and is therefore not appropriate in all situations.

Ring Paxos Ring Paxos [58] is a MultiPaxos variant that decouples control flow from data flow (as in S-Paxos [10]) and that arranges nodes in a chain (as in Chain Replication). As a result, Ring Paxos has the same advantages as S-Paxos and Chain Replication. Like S-Paxos and Mencius, Ring Paxos eliminates some but not all throughput bottlenecks. Acceptors are arranged in a chain, and Phase2B messages are forwarded through the chain much like in Chain Replication. However, the leader is still responsible for sequencing commands and broadcasting to the acceptors and replicas. Ring Paxos also does not optimize reads; reads are processed the same as writes, making the replicas an unnecessary bottleneck in read-heavy workloads.

NoPaxos NoPaxos [51] is a Viewstamped Replication [53] variant that depends on an ordered unreliable multicast (OUM) layer. Each client sends commands to a centralized sequencer that is implemented on a network switch. The sequencer assigns increasing IDs to the commands and broadcasts them to a set of replicas. The replicas speculatively execute commands and reply to clients. In this chapter, we describe how to use proxy leaders to avoid

having a centralized leader. NoPaxos' on-switch sequencer is a hardware based alternative to avoid the bottleneck.

Scalable Agreement In [37], Kapritsos et al. present a protocol similar to Compartmentalized Mencius. The protocol round-robin partitions log entries among a set of replica clusters co-located on a fixed set of machines. Every cluster has $2f + 1$ replicas, with every replica playing the role of a Paxos proposer and acceptor. The protocol can be viewed as an application of compartmentalization. Compartmentalized Mencius extends the protocol with the compartmentalizations described in this chapter, particularly batching and leaderless reads.

SEDA Architecture The SEDA architecture [89] is a server architecture in which functionality is divided into a pipeline of multithreaded modules that communicate with one another using queues. This architecture introduces pipeline parallelism and allows individual components to be scaled up or down to avoid becoming the bottleneck. Our work on decoupling and scaling state machine replication protocols borrows these same ideas, except that we apply them at a fine grain to distributed protocols rather than a single server.

Multithreaded Replication [75] and [8] both propose multithreaded state machine replication protocols. The protocol in [75] is implemented using a combination of actors and the SEDA architecture [89]. A replica's functionality is decoupled into a number of modules, with each module run on its own thread. For example, a MultiPaxos leader has one module to receive messages, one to sequence them, and one to send them. [8] argues for a Mencius-like approach in which each thread has complete functionality (receiving, sequencing, and sending), with slots round-robin partitioned across threads. Multithreaded protocols like these are necessarily decoupled and scale within a single machine. This work is complementary to compartmentalization. Compartmentalization works at the protocol level, while multithreading works on the process level. Both can be applied to a single protocol.

A Family of Leaderless Generalized Protocols In [55], Losa et al. propose a template that can be used to implement state machine replication protocols that are both leaderless and generalized. The template involves a module to compute dependencies between commands and a module to choose and execute commands. The goal of this modularization is to unify existing protocols like EPaxos [62], and Caesar [6]. However, the modularity also introduces decoupling which can lead to performance gains. This is an example of compartmentalization.

Read Leases A common way to optimize reads in MultiPaxos is to grant a lease to the leader [12, 18, 11]. While the leader holds the lease, no other node can become leader. As a result, the leader can perform reads locally without contacting other nodes. Leases assume some degree of clock synchrony, so they are not appropriate in all circumstances. Moreover,

the leader is still a read bottleneck. Raft has a similar optimization that does not require any form of clock synchrony, but the leader is still a read bottleneck [68]. With Paxos Quorum Leases [61], any set of nodes—not just the leader—can hold a lease for a set of objects. These lease holders can read the objects locally. Paxos Quorum Leases assume clock synchrony and are a special case of Paxos Quorum Reads [13] in which read quorums consist of any lease holding node and write quorums consist of any majority that includes all the lease holding nodes. Compartmentalized MultiPaxos does not assume clock synchrony and has no read bottlenecks.

Harmonia Harmonia [98] is a family of state machine replication protocols that leverage specialized hardware—specifically, a specialized network switch—to achieve high throughput and low latency. Like CRAQ, Harmonia is sensitive to data skew. It performs extremely well under low contention, but degrades in performance as contention grows. Harmonia also assumes clock synchrony, whereas Compartmentalized MultiPaxos does not. FLAIR [84] is replication protocol that also leverages specialized hardware, similar to Harmonia.

Sharding In this chapter, we have discussed state machine replication in its most general form. We have not made any assumptions about the nature of the state machines themselves. Because of this, we are not able to decouple the state machine replicas. Every replica must execute every write. This creates a fundamental throughput limit. However, if we are able to divide the state of the state machine into independent shards, then we can further scale the protocols by sharding the state across groups of replicas. For example, in [9], Bezerra et al. discuss how state machine replication protocols can take advantage of sharding.

Low Latency Replication Protocols While compartmentalization increases throughput, it also increases the number of network delays required to get a state machine command executed. For example, starting from a client, MultiPaxos can execute a state machine command and return a response to a client in four network delays, whereas Compartmentalized MultiPaxos requires six. Within a single data center, this translates to a small increase in latency, but when deployed on a wide area network, the latency is increased substantially. Thus, if your goal is to minimize latency, you should choose latency optimized protocols like CURP [69] or SpecPaxos [71] over a compartmentalized protocol.

Chapter 4

Quoracle

MultiPaxos is typically deployed with majority quorums. Given $2f + 1$ acceptors, every set of $f + 1$ acceptors is both a read quorum and a write quorum. Majority quorum systems are widely used in practice because they are easy to understand, because they tolerate an optimal number of faults ($\lfloor \frac{n-1}{2} \rfloor$ with n machines), and because without compartmentalization, the performance of majority quorum systems is good enough. When the leader is a throughput bottleneck, we don't need to optimize the acceptors.

However, after we compartmentalize MultiPaxos, simple majority quorum systems can no longer hide behind bigger bottlenecks. We saw in Section 3.2 that we can eliminate the acceptors as a throughput bottleneck by adopting more sophisticated quorum systems, and we saw that sophisticated quorum systems allowed for huge increases in read throughput by increasing the number of read quorums and leveraging leaderless reads. Thus, sophisticated quorum systems are critical for compartmentalized protocols to achieve high throughput.

The academic literature has proposed quorum systems including Crumbling Walls [70], Trees [2], weighted voting [26, 24], multi-dimensional voting [15], Finite Projective Planes [56], Hierarchies [39], and Paths [64]. These sophisticated quorum systems are a big improvement over simple majority quorum systems, but they have two drawbacks. First, the theory behind these quorum systems ignores many practical considerations such as machine heterogeneity, workload skew, latency, and network load. As we will see in Section 4.3, "theoretically optimal" quorum systems often underperform in practice. Second, understanding the various quorum systems and choosing the one that is optimal for a given workload is difficult and sensitive to workload parameters.

This chapter is a practical re-examination of read-write quorum systems. We revisit the mathematical theory of quorum systems with a pragmatic lens and the ambition to make less-frequently used quorum systems more broadly accessible to the engineering community. More concretely, we make the following contributions:

1. We add a number of practical refinements to the theory of read-write quorum systems (Section 4.1). We extend definitions to accommodate heterogeneous machines and shifting workloads; we introduce the notion of f-resilient strategies to make it easier

to trade off performance for fault tolerance; and we integrate metrics of latency and network load (Section 4.2).

2. We develop a Python library, called Quoracle (Quorum Oracle), that allows users to model, analyze, and optimize read-write quorum systems (Section 4.2). We also provide a heuristic search procedure to find quorum systems that are optimized with respect to a number of user provided objectives and constraints. Given the complex trade-off space, we believe that using an automated assistance library like ours is the only realistic way to find good quorum systems. Quoracle and the associated scripts needed to reproduce this chapter's calculations are available at: `https://github.com/mwhittaker/quoracle`.

3. We perform a case study showing how to use Quoracle to find quorum systems that provide 2× higher throughput or 3× lower latency than naive majority quorums (Section 4.3). These quorum systems can be plugged in to Compartmentalized MultiPaxos to increase the protocol's throughput when acceptors are the bottleneck.

4.1 Definitions

In this section, we present definitions adapted from the existing theory on quorum systems from Naor et al. [64] and Ibaraki et al. [34].

Read-Write Quorum Systems

Given a set $X = \{x_1, \ldots, x_n\}$, a **read-write quorum system** [64] over X is a pair $Q = (R, W)$ where

1. R is a set of subsets of X called **read quorums**,

2. W is a set of subsets of X called **write quorums**, and

3. every read quorum intersects every write quorum. That is, for every $r \in R$ and $w \in W$, $r \cap w \neq \emptyset$.

For example, the majority quorum system over the set $X = \{a, b, c\}$ is $Q_{\text{maj}} = (R, W)$ where $R = W = \{\{a, b\}, \{b, c\}, \{a, c\}\}$. If every read quorum intersects every write quorum, then any superset of a read quorum intersects any superset of a write quorum. Thus, if a set r is a superset of any read quorum in R, we consider r a read quorum as well. Similarly, if a set w is a superset of any write quorum in W, we consider w a write quorum. For example, we consider the set $\{a, b, c\}$ a read and write quorum of Q_{maj} even though the set $\{a, b, c\}$ is not listed explicitly in R or W.

There is a correspondence between sets of quorums and monotone boolean functions [34]. If we associate a boolean variable with every element $x \in X$, then we can express a set of

quorums over X as a monotone boolean expression over these variables. For example, we can represent the set $\{\{a,b\}, \{b,c\}, \{a,c\}\}$ as the expression $(a \wedge b) \vee (b \wedge c) \vee (a \wedge c)$, which we abbreviate as $ab+bc+ac$. Equivalently, we can express the set as $a(b+c)+bc$, $b(a+c)+ac$, or $c(a+b)+ab$.

A read-write quorum system, then, can be expressed as a pair of boolean expressions, one for the read quorums and one for the write quorums. Consider the 2 by 3 grid quorum system $Q_{2\times3}$ over the set $X = \{a,b,c,d,e,f\}$ as shown in Figure 4.1. Every row is a read quorum, and every column is a write quorum. Concretely, $Q_{2\times3} = (abc+def,\ ad+be+cf)$.

(a) Read quorums $abc + def$

(b) Write quorums $ad + be + cf$

Figure 4.1: The 2 by 3 grid quorum system $Q_{2\times3}$.

Given two boolean expressions e_1 and e_2, we say e_1 **dominates** e_2, written $e_2 \leq e_1$, if for any assignment of true and false to the variables in the expressions, if e_2 is true, then e_1 is also true [34]. Intuitively, e_1 dominates e_2 if every quorum in e_2 is also a quorum in e_1. In other words, e_1 has all the quorums of e_2 and potentially more. A read-write quorum system (e_R, e_W) dominates another read quorum system (e'_R, e'_W) if e_R dominates e'_R and e_W dominates e'_W. A read quorum system (e_R, e_W) is **non-dominated** if there does not exist another quorum system $(e'_R, e'_W) \neq (e_R, e_W)$ such that (e'_R, e'_W) dominates (e_R, e_W) [34]. Throughout the chapter, we focus on non-dominated read-write quorum systems because any non-dominated read-write quorum system will be as good or better than any quorum system it dominates.

In practice, X might be a set of machines, a set of locks, a set of memory locations, and so on. In this chapter, we assume that X is a set of machines we call **nodes**. We assume that protocols contact a read quorum of nodes to perform a read and contact a write quorum of nodes to perform a write.

Fault Tolerance

Unfortunately machines fail, and when they do, some quorums become unavailable. For example, if node a from the 2 by 3 grid quorum system $Q_{2\times3}$ fails, then the read quorum $\{a,b,c\}$ and the write quorum $\{a,d\}$ are unavailable. The **read fault tolerance** of a quorum system is the largest number f such that despite the failure of any f nodes, some read quorum is still available [64]. **Write fault tolerance** is defined similarly, and the **fault tolerance** of a quorum system is the minimum of its read and write fault tolerance. For example, the read fault tolerance of $Q_{2\times3}$ is 1 and the write fault tolerance is 2, so the fault tolerance is 1.

Load & Capacity

A protocol uses a **strategy** to decide which quorums to contact when executing reads and writes [64]. Formally, a strategy for a quorum system $Q = (R, W)$ is a pair $\sigma = (\sigma_R, \sigma_W)$ where $\sigma_R : R \to [0, 1]$ and $\sigma_W : W \to [0, 1]$ are discrete probability distributions over the quorums of R and W. $\sigma_R(r)$ is the probability of choosing read quorum r, and $\sigma_W(w)$ is the probability of choosing write quorum w. A **uniform strategy** is one where each quorum is equally likely to be chosen (i.e. $\sigma_R(r) = \frac{1}{|R|}$, $\sigma_W(w) = \frac{1}{|W|}$ for every r and w).

For a node $x \in X$, let $\text{load}_{\sigma_R}(x)$ be the probability that x is chosen by σ_R (i.e. the probability that σ_R chooses a read quorum that contains x). This is called the read load on x. Define $\text{load}_{\sigma_W}(x)$, the write load, similarly. Given a workload with a **read fraction** f_r of reads, the load on x is the probability that x is chosen by strategy σ and is equal to $f_r\text{load}_{\sigma_R}(x) + (1 - f_r)\text{load}_{\sigma_W}(x)$.

The most heavily loaded node is a throughput bottleneck, and its load is what we call the load of the strategy. The **load** of a quorum system is the load of the optimal strategy (i.e. the strategy that achieves the lowest load) [64]. If a quorum system has load L, then the inverse of the load, $\frac{1}{L}$, is called the **capacity** of the quorum system. The capacity of a quorum system is directly proportional to the quorum system's maximum achievable throughput.

For example, consider a 100% read workload (i.e. a workload with read fraction $f_r = 1$) and consider again the grid quorum system $Q_{2\times3}$ in Figure 4.1. The optimal strategy is a uniform strategy that selects both read quorums equally likely. Thus, the load of $Q_{2\times3}$ is $\frac{1}{2}$, and its capacity is 2. If every node can process α commands per second, then the quorum system can process 2α commands per second in aggregate. Alternatively, consider a 100% write workload with a read fraction $f_r = 0$. The optimal strategy is again uniform. Because there are three write quorums, the load is $\frac{1}{3}$, and the capacity is 3. The quorum system can process 3α commands per second under this workload. Finally, with $f_r = \frac{1}{2}$ (i.e. a workload with 50% reads and 50% writes), the quorum system's capacity is $\frac{12}{5}$.

4.2 Practical Refinements in Quoracle

In this section, we augment the theory of read-write quorum systems with a number of practical considerations and demonstrate their use in our Python library Quoracle.

Quorum Systems, Capacity, Fault Tolerance

Quoracle allows users to form arbitrary read-write quorum systems and compute their capacity and fault tolerance. For example, in Figure 4.2, we construct and analyze the majority quorum system on nodes $\{a, b, c\}$. As in Section 4.1, read-write quorum systems are constructed from a set of read or write quorums expressed as a boolean expression over the set of nodes.

```
a, b, c = Node('a'), Node('b'), Node('c')
majority = QuorumSystem(reads=a*b + b*c + a*c)
print(majority.fault_tolerance())         # 1
print(majority.load(read_fraction=1))     # 2/3
print(majority.capacity(read_fraction=1)) # 3/2
```

Figure 4.2: Quorum systems, capacity, and fault tolerance.

Note that the user only has to specify one set of quorums rather than both because we automatically construct the optimal set of complementary quorums using the existing body of literature that relates read-write quorum systems to monotone boolean functions [34]. Specifically, given a boolean expression e, the **dual** of e, denoted $\text{dual}(e)$ is the expression formed by swapping logical and ($\wedge$) with logical or ($\vee$) in e. For example, $\text{dual}(ab) = a + b$, $\text{dual}(a + b) = ab$, and $\text{dual}(a(b + c) + de) = (a + bc)(d + e)$. As described in [34], given a boolean expression e_R representing a set of read quorums over a set X, the optimal set of complementary write quorums is $e_W = \text{dual}(e_R)$. Formally, (e_R, e_W) is non-dominated. Similarly, given an expression e_W representing a set of write quorums, the optimal set of complementary read quorums is $e_R = \text{dual}(e_W)$. This is how Quoracle computes write quorums when only a set of read quorums is given (and vice versa).

Quoracle computes the load of a quorum system using linear programming [64]. Specifically, given a read-write quorum system $Q = (R, W)$ over a set X with read fraction f_r, we introduce a load variable L, a variable p_r for every $r \in R$, and a variable p_w for every $w \in W$. The linear program computes the optimal strategy $\sigma^* = (\sigma^*_R, \sigma^*_W)$. L represents the load of σ^*, p_r represents $\sigma^*_R(r)$, and p_w represents $\sigma^*_W(w)$. The linear program minimizes L with the following constraints. First, $0 \leq p_r, p_w \leq 1$ for every p_r and p_w. Second, $\sum_{r \in R} p_r = 1$ and $\sum_{w \in W} p_w = 1$. These two constraints ensure that strategies σ^*_R and σ^*_W are valid probability distributions. Third, for every $x \in X$,

$$f_r \left(\sum_{\{r \in R \,|\, x \in r\}} p_r \right) + (1 - f_r) \left(\sum_{\{w \in W \,|\, x \in w\}} p_w \right) \leq L$$

This constraint ensures that the load on node x is less than or equal to L.

Quoracle computes the fault tolerance of a quorum system using integer programming. First, we form an integer program to compute read fault tolerance. We introduce a variable $v_x \in \{0, 1\}$ for every $x \in X$. Intuitively, if $v_x = 1$, it means node x has failed, and if $v_x = 0$, it means node x is alive. We minimize $\sum_{x \in X} v_x$ with the constraint that for every $r \in R$, $\sum_{x \in r} v_x \geq 1$. By minimizing $\sum_{x \in X} v_x$, we try to fail as few nodes as possible. The constraint $\sum_{x \in r} v_x \geq 1$ ensures that at least one node from r has failed. We then compute the read fault tolerance as $f = (\sum_{x \in X} v_x) - 1$. $f + 1$ is the minimum number of nodes we can fail to eliminate all read quorums, so the quorum system can tolerate as many as f failures. We

solve for the write fault tolerance in the same way. The fault tolerance is the minimum of the read and write fault tolerance.

Heterogeneous Nodes

Quorum system theory implicitly assumes that all nodes are equal. In reality, nodes are often heterogeneous. Some are fast, and some are slow. Moreover, nodes can often process more reads per second than writes per second. We revise the theory by associating every node x with its read and write capacity, i.e. the maximum number of reads and writes the node can process per second. We redefine the read load imposed by a strategy $\sigma = (\sigma_R, \sigma_W)$ on a node x as the probability that σ_R chooses x divided by the read capacity of x. We redefine the write load similarly. By normalizing a node's load with its capacity, we get a more intuitive definition of a quorum system's capacity. Now, the capacity of a quorum system is the maximum throughput that it can support.

Quoracle allows users to annotate nodes with read and write capacities. For example, in Figure 4.3, we construct a 2 by 2 grid quorum system where nodes a and b can process 100 writes per second, but nodes c and d can only process 50 writes per second. We also specify that every node can process reads twice as fast as writes. With a read fraction of 1, the quorum system has a capacity of 300 commands per second using a strategy that picks the read quorum $\{a, b\}$ twice as often as the read quorum $\{c, d\}$. As we decrease the fraction of reads, the capacity decreases since the nodes process reads faster than writes.

```
a = Node('a', write_cap=100, read_cap=200)
b = Node('b', write_cap=100, read_cap=200)
c = Node('c', write_cap=50, read_cap=100)
d = Node('d', write_cap=50, read_cap=100)
grid = QuorumSystem(reads=a*b + c*d)
print(grid.capacity(read_fraction=1))   # 300
print(grid.capacity(read_fraction=0.5)) # 200
print(grid.capacity(read_fraction=0))   # 100
```

Figure 4.3: Heterogeneous nodes with different capacities.

To compute the load and capacity of a read-write quorum systems with different read and write capacities, Quoracle modifies its linear program by normalizing every node's load by its capacity. Specifically, for every node $x \in X$, it uses the following constraint where $\text{cap}_R(x)$ and $\text{cap}_W(x)$ are the read and write capacities of node x:

$$\left(\frac{f_r}{\text{cap}_R(x)} \sum_{\{r \in R \,|\, x \in r\}} p_r \right) + \left(\frac{1 - f_r}{\text{cap}_W(x)} \sum_{\{w \in W \,|\, x \in w\}} p_w \right) \leq L$$

Workload Distributions

Capacity is defined with respect to a fixed read and write fraction, but in reality, workloads skew. To accommodate workload skew, we consider a discrete probability distribution over a set of read fractions and redefine the capacity of a quorum system to be the capacity of the strategy σ that maximizes the expected capacity with respect to the distribution. For example, in Figure 4.4, we construct the quorum system with read quorums $ac + bd$, and we define a workload that has 0% reads $\frac{10}{18}$th of the time, 25% reads $\frac{4}{18}$th of the time, and so on. In Figure 4.4, we see the optimal strategy σ has an expected capacity of 159 commands per second.

```
grid = QuorumSystem(reads=a*c + b*d)
fr = {0.00: 10/18, 0.25: 4/18, 0.50: 2/18,
      0.75:  1/18, 1.00: 1/18}
sigma = grid.strategy(read_fraction=fr)
print(sigma.capacity(read_fraction=fr)) # 159
```

Figure 4.4: A distribution of read fractions.

In Figure 4.5, we plot strategy σ's capacity as a function of read fraction. We also plot the capacities of strategies $\sigma_{0.0}$, $\sigma_{0.25}$, $\sigma_{0.50}$, $\sigma_{0.75}$, and $\sigma_{1.0}$ where σ_{f_r} is the strategy optimized for a fixed workload with a read fraction of f_r. We see that strategy σ does not always achieve the maximum capacity for any *individual* read fraction, but it achieves the best expected capacity across the distribution.

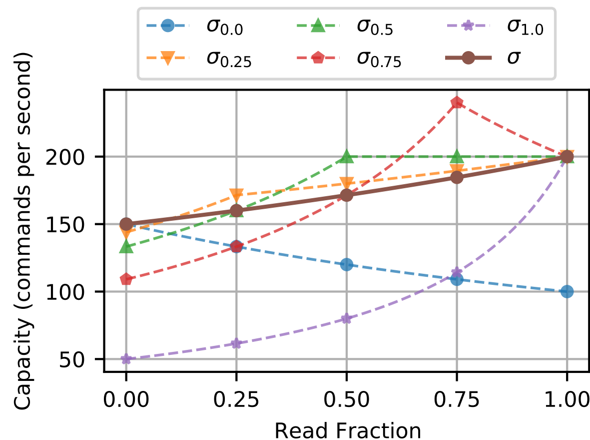


Figure 4.5: Strategy capacities with respect to read fraction

Note that strategy σ performs well across all workloads drawn from the distribution without having to know the current read fraction. Alternatively, if we are able to monitor

the workload and deduce the current read fraction, we can pre-compute a set of strategies that are optimized for various read fractions and dynamically select the one that is best for the current workload.

To compute the load and capacity of a read-write quorum system with a distribution of read fractions, Quoracle again modifies its linear program. Rather than minimizing a single load variable L, we have one load variable L_{f_r} for every possible value of f_r and minimize their sum, weighted according to their distribution. For every node $x \in X$ and every value of read fraction f_r, the linear program has the constraint:

$$\left(\frac{f_r}{\text{cap}_R(x)} \sum_{\{r \in R \,|\, x \in r\}} p_r \right) + \left(\frac{1 - f_r}{\text{cap}_W(x)} \sum_{\{w \in W \,|\, x \in w\}} p_w \right) \leq L_{f_r}$$

f-resilient Strategies

Many protocols that deploy read-write quorum systems actually contact more nodes than is strictly necessary when executing a read or a write. Rather than contacting a quorum to perform a read or write, these protocols contact *every* node. Contacting every node leads to suboptimal capacity, but it is less sensitive to stragglers and node failures. For example, if we contact only a quorum of nodes and one of the nodes in the quorum fails, then we have to detect the failure and contact a different quorum. This can be slow and costly. Typically, industry practitioners have chosen between these two extremes: either send messages to *every node* or send messages to the *bare minimum number of nodes* (i.e. a quorum) [80, 58, 50, 36, 40, 11]. We introduce the notion of f-resilient strategies to show that this is not a binary decision, but rather a continuous trade-off.

Given a quorum system (R, W), we say a read quorum $r \in R$ is f-resilient for some integer f if despite removing any f nodes from r, r is still a read quorum. We define f-resilience for write quorums similarly. We say a strategy σ is **f-resilient** if it only selects f-resilient read and write quorums. An f-resilient strategy can tolerate any f failures or stragglers. The value of f captures the continuous trade-off between capacity and resilience. As we increase f, we decrease capacity but increase resilience.

Quoracle allows users to compute optimal f-resilient strategies and their corresponding capacities. For example, in Figure 4.6, we compute the f-resilient capacity of a grid quorum system for $f = 0$ and $f = 1$. Its 0-resilient capacity is 300, but its 1-resilient capacity is only 100. We then compute the f-resilient capacities for the "read 2, write 3" quorum system. For this quorum system, every set of two nodes is a read quorum, and every set of three nodes is a write quorum. This quorum system has the same 0-resilient capacity as the grid but a higher 1-resilient capacity, showing that some quorum systems are naturally more resilient than others.

Quoracle computes f-resilient quorums using a brute-force backtracking algorithm with pruning. Given a set of nodes X, Quoracle enumerates every subset of X and checks whether it is an f-resilient quorum. However, once an f-resilient quorum is found, all supersets of the quorum are pruned from consideration.

```
grid = QuorumSystem(reads=a*b + c*d)
print(grid.capacity(read_fraction=1, f=0)) # 300
print(grid.capacity(read_fraction=1, f=1)) # 100
read2 = QuorumSystem(reads=choose(2, [a,b,c,d]))
print(read2.capacity(read_fraction=1, f=0)) # 300
print(read2.capacity(read_fraction=1, f=1)) # 200
```

Figure 4.6: 0-resilient and 1-resilient strategies.

Latency and Network Load

Quorum system theory focuses on capacity and fault tolerance. We introduce two new practically important metrics. First, we introduce **latency**. We associate every node with a latency that represents the time required to contact the node. The latency of a quorum q is the time required to form a quorum of responses after contacting the nodes in q. The latency of a strategy is the expected latency of the quorums that it selects. The lower the latency, the better. Note that if a quorum is f-resilient, we only need to collect responses from at most all but f of the nodes in order to form a quorum, so the latency of a quorum can be less than the latency required to hear back from *every* node in the quorum.

Second, we introduce **network load**. When a protocol executes a read or write, it sends messages over the network to every node in a quorum, so as the sizes of quorums increase, the number of network messages increases. The network load of a strategy is the expected size of the quorums it chooses. The lower the network load, the better.

In isolation, optimizing for latency or network load is trivial, but balancing capacity, fault tolerance, latency, and network load simultaneously is very complex. Quoracle allows users to find strategies that are optimal with respect to capacity, latency, or network load with constraints on the other metrics. For example, in Figure 4.7, we specify the latencies of the nodes in our 2 by 2 grid quorum system and then find the latency optimal strategy with a capacity no less than 150 and with a network load of at most 2. The optimal strategy has a latency of 3 seconds.

Quoracle again uses linear programming to optimize latency and network load. The latency of a quorum system is computed as follows where latency(r) and latency(w) are the latencies of read quorum r and write quorum w:

$$f_r \left(\sum_{r \in R} p_r \cdot \text{latency}(r) \right) + (1 - f_r) \left(\sum_{w \in W} p_w \cdot \text{latency}(w) \right)$$

The network load is computed as

$$f_r \left(\sum_{r \in R} p_r \cdot |r| \right) + (1 - f_r) \left(\sum_{w \in W} p_w \cdot |w| \right)$$

```
a = Node('a', write_cap=100, read_cap=200, latency=4)
b = Node('b', write_cap=100, read_cap=200, latency=4)
c = Node('c', write_cap=50, read_cap=100, latency=1)
d = Node('d', write_cap=50, read_cap=100, latency=1)
grid = QuorumSystem(reads=a*b + c*d)
sigma = grid.strategy(read_fraction = 1,
                      optimize = 'latency',
                      capacity_limit = 150,
                      network_limit = 2)
print(sigma.latency(read_fraction=1)) # 3 seconds
```

Figure 4.7: Finding a latency-optimal strategy with capacity and network load constraints.

Note that in reality, the relationships between load, latency, and network load are complex. For example, as the load on a node increases, the latencies of the requests sent to it increase. Moreover, the clients that communicate with the nodes in a quorum system may experience different latencies based on where they are physically located. We leave these complexities to future work.

Quorum System Search

Thus far, we have demonstrated how Quoracle makes it easy to model, analyze, and optimize a *specific hand-chosen* quorum system. Quoracle also implements a heuristic based search procedure to find good quorum systems. For example, in Figure 4.8, we search for a quorum system over the nodes $\{a, b, c, d\}$ optimized for latency with a capacity of at least 150 and a network load of at most 2. The search procedure returns the quorum system with read quorums $a + b + c + d$ and write quorums $abcd$, and with the read strategy that picks c one third of the time and d two thirds of the time.

Given a list of expressions $\bar{e} = e_1, \ldots, e_n$, let $\text{choose}(k; \bar{e})$ be the disjunction of the conjunction of every set of k expressions in $\bar{e}$. For example, $\text{choose}(2; a, b, c) = ab + ac + bc$, and $\text{choose}(1; a, b, c) = a + b + c$. Given a boolean expression e representing a set of quorums, we say e is **duplicate free** if e can be expressed using logical or, logical and, and choose with every variable in e appearing exactly once. For example $a + bc$ is duplicate free. $ab + ac = a(b + c)$ is duplicate free. $ab + ac + bc = \text{choose}(2; a, b, c)$ is duplicate free. $ab + ace + de + dcb$ is *not* duplicate free.

Our search procedure exhaustively searches the space of all quorum systems that have read quorums expressible by a duplicate free expression. The search procedure heuristically explores simpler expressions first. Specifically, it enumerates expressions in increasing order of their depth when represented as an abstract syntax tree. Because the search space is enormous, users can specify a timeout.

```
qs, sigma = search(nodes = [a, b, c, d],
                   read_fraction = 1,
                   optimize = 'latency',
                   capacity_limit = 150,
                   network_limit = 2)
print(qs) # reads=a+b+c+d, writes=a*b*c*d
print(sigma) # c: 1/3, d: 2/3
print(sigma.latency(read_fraction=1)) # 1 second
print(sigma.capacity(read_fraction=1)) # 150
```

Figure 4.8: Searching the space of quorum systems.

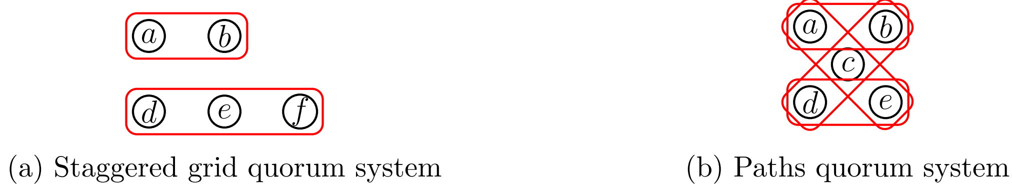
4.3 Case Study

In this section, we present a hypothetical case study that demonstrates how to use Quoracle in a realistic setting. Assume we have five nodes. Nodes a, c, and e can process 2,000 writes per second, while nodes b and d can only process 1,000 writes per second. All nodes process reads twice as fast as writes. Nodes a and b have a latency of 1 second; nodes c, d, and e have latencies of 3, 4, and 5 seconds. We observe a workload with roughly equal amounts of reads and writes with a slight skew towards being read heavy. In Figure 4.9, we use Quoracle to model the nodes and workload distribution.

Assume we have already deployed a majority quorum system with a uniform strategy, which has a capacity of 2,292 commands per second. We want to find a more load optimal quorum system. We consider three candidates. The first is the majority quorum system. The second is a staggered grid quorum system, illustrated in Figure 4.10a. The third is a quorum system based on paths through a two-dimensional grid illustrated in Figure 4.10b. This quorum system has theoretically optimal capacity [64]. In Figure 4.11, we construct these three quorum systems and print their capacities.

```
a = Node('a', write_cap=2000, read_cap=4000, latency=1)
b = Node('b', write_cap=1000, read_cap=2000, latency=1)
c = Node('c', write_cap=2000, read_cap=4000, latency=3)
d = Node('d', write_cap=1000, read_cap=2000, latency=4)
e = Node('e', write_cap=2000, read_cap=4000, latency=5)
fr = {0.9:  10/470, 0.8:  20/470, 0.7: 100/470,
      0.6: 100/470, 0.5: 100/470, 0.4:  60/470,
      0.3:  30/470, 0.2:  30/470, 0.1:  20/470}
```

Figure 4.9: Nodes and workload distribution.



(a) Staggered grid quorum system

(b) Paths quorum system

Figure 4.10: The read quorums of the staggered grid and paths quorum systems. The optimal set of complementary write quorums is chosen automatically.

```
maj = QuorumSystem(reads=majority([a, b, c, d, e]))
grid = QuorumSystem(reads=a*b + c*d*e)
paths = QuorumSystem(reads=a*b + a*c*e + d*e + d*c*b)
print(maj.capacity(reads_fraction=fr))   # 3,667
print(grid.capacity(reads_fraction=fr))  # 4,200
print(paths.capacity(reads_fraction=fr)) # 4,125
```

Figure 4.11: Quorum systems and their capacities.

The capacities are 3,667, 4,200, and 4,125 commands per second respectively, making the grid quorum system the most attractive. However, the grid quorum system is not necessarily optimal. In Figure 4.12, we perform a search for a quorum system optimized for capacity that is tolerant to one failure. The search takes 7 seconds on a laptop.

```
qs, sigma = search(nodes=[a, b, c, d, e],
                   fault_tolerance=1,
                   read_fraction=fr)
print(qs.capacity(read_fraction=fr)) # 5,005
```

Figure 4.12: Searching for a load-optimal quorum system.

The search procedure finds the quorum system with read quorums $(c+bd)(a+e)$ which has a capacity of 5,005 commands per second. This is 1.19× better than the grid quorum system, and 2.18× better than the majority quorum system with a naive uniform strategy. Assume hypothetically that we deploy this strategy to production. Months later, we introduce a component into our system that bottlenecks our throughput at 2,000 commands per second. Now, any capacity over 2,000 is wasted, so we search for a quorum system optimized for latency with a capacity of at least 2,000. We again consider our three quorum systems in Figure 4.13.

```
for qs in [maj, grid, paths]:
    print(qs.latency(read_fraction=fr,
                     optimize='latency',
                     capacity_limit=2000))
```

Figure 4.13: Latencies with a capacity constraint.

The quorum systems have latencies of 3.24, 1.95, and 2.43 seconds respectively, making the grid quorum system the most attractive. We again perform a search and find the quorum system with read quorums $ab + acde + bcde$ achieves a latency of 1.48 seconds. This is 1.32× better than the grid and 3.04× better than a naive uniform strategy over a majority quorum system. The search again completes in 7 seconds. We hypothetically deploy this quorum system to production.

4.4 Lessons Learned

Naive Majority Quorums Underperform

Industry practitioners often use majority quorums because they are simple and have strong fault tolerance. Our case study shows that majority quorum systems with uniform strategies almost always underperform more sophisticated quorum systems in terms of capacity, latency, and network load. In Figure 4.14, we plot a stacked histogram of the throughput that every node in a majority quorum system obtains using a naive uniform strategy, with throughput broken down by quorums. We contrast this in Figure 4.15 with the strategy found in Figure 4.12. The sophisticated quorum system assigns more work to machines with higher capacities, leading to a 2.18× increase in aggregate throughput. Moreover, the performance benefits of more sophisticated quorum systems do not sacrifice correctness, unlike alternatives like sloppy quorums [19].

"Optimal" Is Not Always Best.

There is a large body of research on constructing "optimal" quorum systems [24, 56, 2, 39, 16, 64, 70]. For example, the paths quorum system is theoretically optimal, but in our case study, it has lower capacity and higher latency than the simpler grid quorum system. There are two reasons for this mismatch between theoretical and practical optimality. First, existing quorum system theory does not account for node heterogeneity and workload skew. Second, these quorum systems are only optimal in the limit, as the number of nodes tends to infinity.

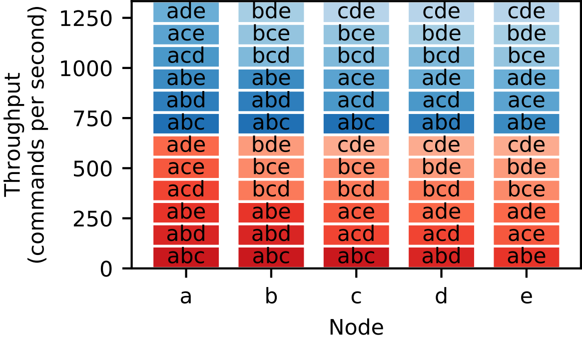


Figure 4.14: A stacked histogram of the throughput of a simple majority quorum system with a naive uniform strategy. Write quorums are in blue, and read quorums are in red.

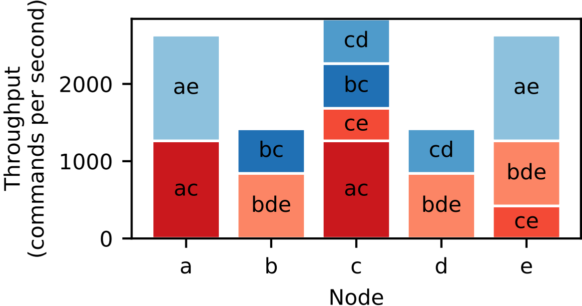


Figure 4.15: A stacked histogram of the throughput of the quorum system found by our heuristic search (i.e., the quorum system with read quorums $(c + bd)(a + e)$).

The Trade-Off Space Is Complex

Constructing a quorum system of homogeneous nodes that is optimal in the limit for a fixed workload is difficult but doable. When nodes operate at different speeds and workloads skew, finding an optimal quorum system that satisfies constraints on capacity, fault tolerance, latency, and network load becomes nearly impossible to do by hand. Moreover, small perturbations in any of these parameters can change the landscape of the optimal quorum systems. In our case study, for example, the search procedure finds two different quorum systems when optimizing for load and when optimizing for latency. We believe that using an automated assistance library like ours is the only realistic way to find good quorum systems.

Chapter 5

Bipartisan Paxos

In Chapter 3, we saw how to use compartmentalization to make state machine replication protocols faster without making them more complicated. Now, we show how to use compartmentalization to make complicated state machine replication protocols easier to understand without making them slower.

There is a family of *generalized multi-leader* state machine replication protocols, including EPaxos [62], Caesar [6], and Atlas [21]. These protocols are **multi-leader**, which means that they have multiple simultaneous leaders. This avoids having a single bottlenecked leader. They are also **generalized** [42, 55], which means that every replica executes non-commuting commands in the exact same order, but are otherwise free to execute commuting commands in any order. In other words, these protocols execute a partially ordered graph of commands rather than a totally ordered log of commands.

These generalized multi-leader protocols are extremely complicated. Paxos has a well known reputation for being complex [44, 87, 68], and these generalized multi-leader protocols are significantly more complex than that. They require a strong understanding of more sophisticated Paxos variants like Fast Paxos [41] and are overall less intuitive and more nuanced. It is hard to measure this complexity precisely, but there are indications that the protocols are complicated. EPaxos, for example, had several bugs go undiscovered for years despite the popularity of the protocol [81].

This complexity has negative consequences in industry and academia. Generalized multi-leader protocols have little to no industry adoption. We postulate that this is largely due to their complexity. The engineers in [12] explain that implementing a state machine replication protocol requires making many small changes to the protocol to match the environment in which it is deployed. Making these changes without a strong understanding of the protocol is infeasible. Academically, it is challenging to understand the novelty of each protocol because the protocols have not been cleanly factored. The protocols are all very similar, yet they also have lots of small differences between them, some novel and some inconsequential. Without a clean factoring of the protocols' designs, it is difficult to understand where each protocol is novel. This makes it difficult to extend the protocols with further innovations, as it is unclear which ideas have already been explored and which have yet to be examined.

In this chapter, we use compartmentalization as a tool to understand these protocols. Rather than presenting a monolithic, nuanced, and highly optimized protocol, we present a fully compartmentalized and completely unoptimized generalized multi-leader protocol that is designed to be simple to understand. We then slowly reassemble the components and introduce optimizations to recreate existing protocols like EPaxos, Caesar, and Atlas. In doing so, we create a framework for understanding generalized multi-leader protocols.

The tutorial has four parts, and in each part, we introduce a new protocol. First, we present a fully compartmentalized generalized multi-leader protocol, which we called **Simple BPaxos** (Section 5.2). Simple BPaxos sacrifices performance for simplicity and is designed with the sole goal of being easy to understand. Simple BPaxos is the kernel from which all other generalized multi-leader protocols can be constructed. It encapsulates all the mechanisms and invariants that are common to the other protocols.

Second, we introduce a purely pedagogical protocol called **Fast BPaxos** (Section 5.4). Fast BPaxos achieves higher performance than Simple BPaxos, but it is unsafe. The protocol does not properly implement state machine replication. Why study a broken protocol? Because understanding why Fast BPaxos does *not* work leads to a fundamental insight on why other protocols do. Specifically, we discover that generalized multi-leader protocols encounter a fundamental tension between agreeing on commands and ordering commands. In isolation, reaching consensus on what command to execute is easy, and determining how the command should be ordered with respect to other commands is easy, but doing both at the same time is hard. This is where the tension arises. The way in which a protocol handles this tension is its key distinguishing feature. We taxonomize the protocols into those that avoid the tension and those that resolve the tension.

Third, we introduce **Unanimous BPaxos**, a simple *tension avoiding* protocol (Section 5.5). We describe how tension avoiding protocols carefully enlarge quorum sizes to sidestep the tension. We also explain how Basic EPaxos [62] and Atlas [21] can be expressed as optimized variants of Unanimous BPaxos.

Fourth, we introduce **Majority Commit BPaxos**, a *tension resolving* protocol (Section 5.6). We describe how tension resolving protocols perform detective work to resolve the tension without enlarging quorum sizes. We also discuss the relationship between Majority Commit BPaxos and the tension resolving protocols EPaxos [60] and Caesar [6]. A full taxonomy of these protocols and other related protocols is shown in Figure 5.13 at the end of this chapter.

5.1 Conflict Graphs

Defining Conflict Graphs

By totally ordering state machine commands into a log, state machine replication protocols like MultiPaxos ensure that *every* replica executes *every* command in *exactly the same order*. This is a simple way to ensure that replicas are always in sync, but it is sometimes unnec-

essary [42]. For example, consider the log shown at the top of Figure 5.1. The command a=2 (i.e. set the value of variable a to 2) is chosen in log entry 1, and the command b=1 is chosen in log entry 2. With MultiPaxos, every replica would execute these two commands in exactly the same order, but this is not necessary because the commands commute. It is safe for some replicas to execute a=2 before b=1 while other replicas execute b=1 before a=2. The execution order of the two commands has no effect on the final state of the state machine, so they can be safely reordered, as shown in Figure 5.1.

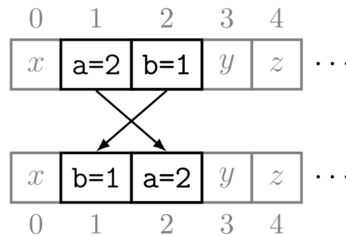


Figure 5.1: If two commands commute, replicas can safely execute them in either order.

More formally, we say two commands x and y **conflict** if there exists a state in which executing x and then y does not produce the same responses or final state as executing y and then x. We say two commands **commute** if they do not conflict. If two commands conflict (e.g., a=1 and a=2), then they need to be executed by every state machine replica in the same order. But, if two commands commute (e.g., a=2 and b=1), then they do *not* need to be totally ordered. State machine replicas can execute them in either order.

Generalized Multi-leader state machine replication protocols like EPaxos, Caesar, Atlas, and all the BPaxos variants presented in this chapter take advantage of command commutativity. Rather than totally ordering commands into a log, these protocols *partially* order commands into a directed graph such that every pair of conflicting commands has an edge between them. We call these graphs **conflict graphs**. An example log and corresponding conflict graph is illustrated in Figure 5.2. A log consists of a number log entries, and every log entry has a unique log index (e.g., 4). A conflict graph consists of a number of **vertices**, and every vertex has a unique **vertex id** (e.g., v_4).

Moreover, a vertex v can have directed edges to other vertices. These are called the **dependencies** of v, denoted deps(v). For example, if vertex v_i depends on vertex v_j, then there is an edge from v_i to v_j. Note that if a pair of commands conflict, then they must have an edge between them. This ensures that every replica executes the two commands in the same order. For example in Figure 5.2, the commands a=b (v_0) and a=2 (v_1) conflict, so they have an edge between them. If two commands commute, then they do not have an edge between them. This allows replicas to execute the commands in either oder. For example, the commands a=2 (v_1) and b=1 (v_2) commute, so there is no edge between them. Finally note that some conflicting commands (e.g., b=a (v_3) and a=3 (v_4)) have edges in

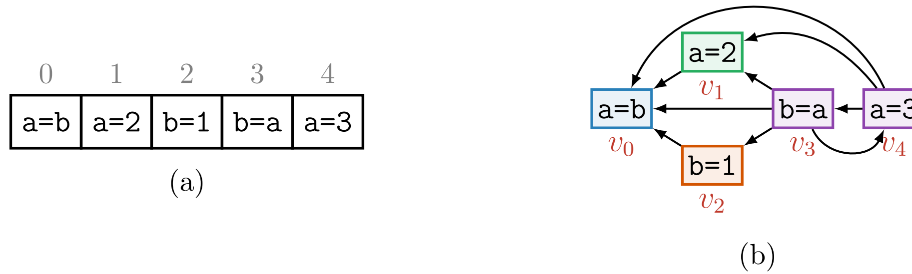


Figure 5.2: A log and corresponding conflict graph.

both directions, forming a cycle. Ideally, conflict graphs would be acyclic, but cycles are sometimes unavoidable. The reason for this will become clear soon.

Executing Conflict Graphs

We now explain how to execute a static conflict graph. In the next subsection, we explain how to execute a dynamic conflict graph that grows over time. Replicas execute logs in prefix order. Replicas execute conflict graphs in reverse topological order, one strongly connected component at a time. The order of executing commands within a strongly connected component is not important, but every replica must choose the same order. For example, replicas can execute commands within a component sorted by their vertex id. The conflict graph in Figure 5.2 has four strongly connected components, each shaded a different color. Vertices v_0, v_1, and v_2 are each in their own components, and commands v_3 and v_4 are in their own component. Replicas execute these four strongly connected components in reverse topological order as follows:

- First, replicas execute `a=b` (v_0).
- Next, replicas either execute `a=2` (v_1) then `b=1` (v_2) or `b=1` (v_2) then `a=2` (v_1). There are no edges between vertex v_1 and vertex v_2, so every replica can execute the two vertices in either order.
- Finally, replicas execute `b=a` (v_3) and `a=3` (v_4) in some arbitrary but fixed order. For example, if replicas execute commands sorted by their vertex ids, then the replicas would all execute v_3 and then v_4.

Executing commands in this way, state machine replicas are guaranteed to remain in sync. Every replica executes conflicting commands in the same order, but are free to execute commuting commands in any order.

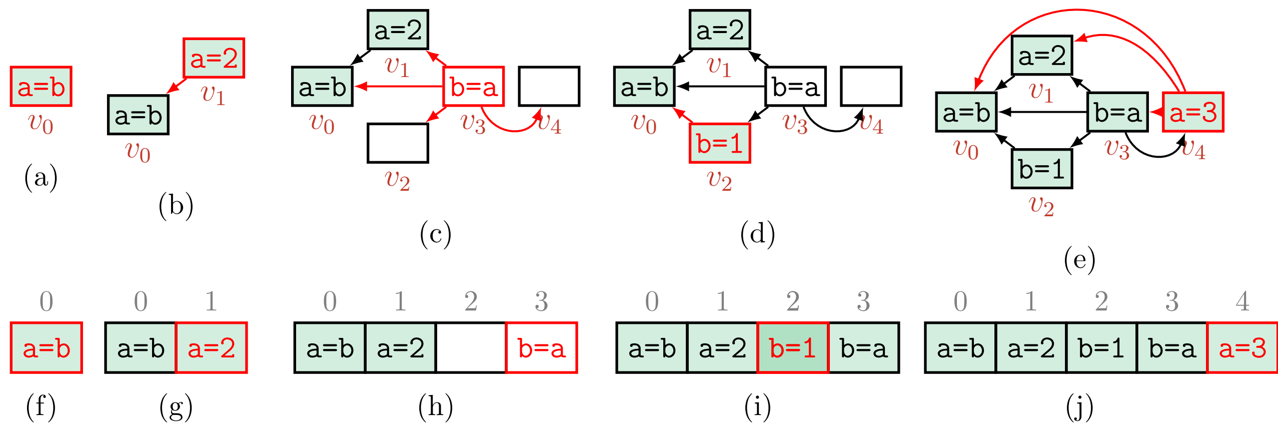


Figure 5.3: In subfigures (a) - (e), we see a conflict graph constructed over time. The most recently chosen vertex is drawn in red. The executed commands are shaded green. (a) The command `a=b` is chosen in vertex v_0 without any dependencies. The command is executed immediately. (b) The command `a=2` is chosen in vertex v_1 with a dependency on v_0. The command is executed immediately. (c) The command `b=a` is chosen in vertex v_3 with dependencies on v_0, v_1, v_2, and v_4. No commands have been chosen in v_2 and v_4 yet, so v_3 cannot be executed. (d) The command `b=1` is chosen in vertex v_2 with a dependency on v_0. The command is executed immediately. (e) The command `a=3` is chosen in vertex v_4 with dependencies on v_0, v_1, and v_3. Now v_3 and v_4 are executed. In subfigures (f) - (j), we see an analogous execution for a log.

Constructing Conflict Graphs

In the previous subsection, we explained how to execute a static conflict graph. In reality, graphs are dynamic and grow over time. MultiPaxos constructs one log entry at a time. It uses one instance of consensus for every log entry i to choose which command should be placed in log entry i. Analogously, generalized multi-leader protocols construct a conflict graph one vertex at a time. They use one instance of consensus for every vertex v to choose which command should be placed in vertex v and what dependencies, or outbound edges, v should have.

In Figure 5.3, we illustrate an example execution of how the conflict graph from Figure 5.2 could be constructed over time. Figure 5.3 also shows an analogous execution in which a log is constructed over time. Note that a vertex v can be chosen with dependencies $\text{deps}(v)$ before every vertex in $\text{deps}(v)$ has itself been chosen. For example in Figure 5.3c, v_3 is chosen with $\text{deps}(v_3) = \{v_0, v_1, v_2, v_4\}$ before vertices v_2 and v_4 are chosen. This is analogous to how a command is chosen in log entry 3 in Figure 5.3h before a command is chosen in entry 2.

A summary of the differences between logs and graphs is given in Table 5.1.

Table 5.1: The differences between protocols like MultiPaxos and Raft that organize commands in logs and protocols like EPaxos, Caesar, and Atlas that organize commands in graphs.

	Logs	Graphs
data structure	log	conflict graph
	log entry	vertex
	log index (e.g., 4)	vertex id (e.g., v_4)
	total order	partial order
execution order	log order	reverse topological order
what's chosen?	commands	commands & dependencies

Two Key Invariants

Protocols like EPaxos, Caesar, Atlas, and the BPaxos protocols in this chapter all differ in how they assign commands to vertices, how they compute dependencies, how they implement consensus, and so on. Despite the differences, all the protocols construct conflict graphs one vertex at a time, choosing a command and a set of dependencies $(x, \text{deps}(v))$ for every vertex v. The protocols all rely on the following two key invariants for correctness. We call these the **consensus invariant** (Invariant 1) and the **dependency invariant** (Invariant 2).

Invariant 1 (Consensus Invariant)**.** Consensus is implemented for every vertex v. That is, at most one value $(x, \text{deps}(v))$ is chosen for every vertex v.

Invariant 2 (Dependency Invariant)**.** If $(x, \text{deps}(v_x))$ is chosen in vertex v_x and $(y, \text{deps}(v_y))$ is chosen in instance v_y, and if x and y conflict, then either $v_x \in \text{deps}(v_y)$ or $v_y \in \text{deps}(v_x)$ or both. That is, if two chosen commands conflict, there is an edge between them.

The consensus invariant ensures that replicas always agree on the state of the conflict graph. It makes it impossible, for example, for two replicas to disagree on which command is in a vertex or disagree on what dependencies a vertex has. The dependency invariant ensures that replicas execute conflicting commands in the same order but does not require that replicas execute commuting commands in the same order. These two invariants are sufficient to ensure linearizable execution. Intuitively, the history of command execution is equivalent to a serial history following any reverse topological ordering of the conflict graph. In fact, replicas literally do execute commands serially according to one of the reverse topological orderings. For a more formal proof, refer to [42] and [60].

5.2 Simple BPaxos

In this section, we introduce **Simple BPaxos**, an inefficient protocol that is designed to be easy to understand. By understanding Simple BPaxos, we will understand of the core mechanisms and invariants that are common to all generalized multi-leader protocols.

Overview

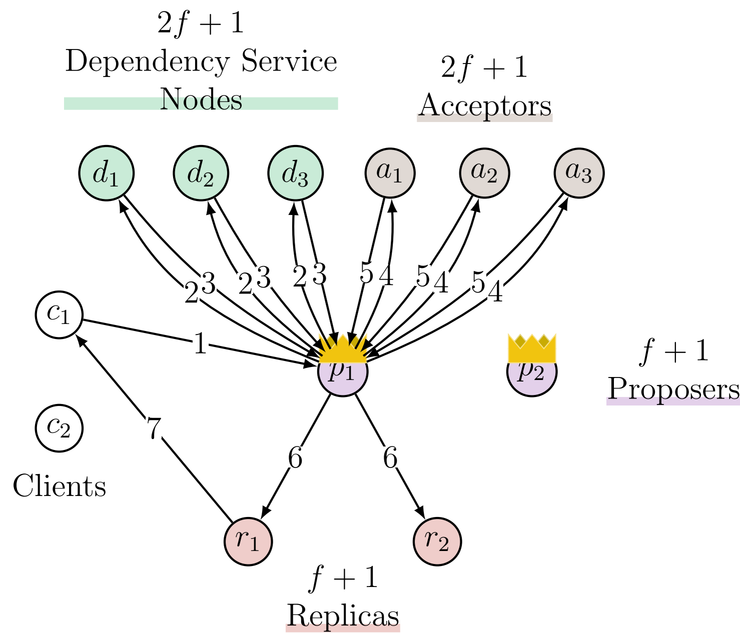


Figure 5.4: An example execution of Simple BPaxos ($f = 1$).

As illustrated in Figure 5.4, a Simple BPaxos deployment consists of a number of clients, a set of at least $f+1$ Paxos proposers, a set of $2f+1$ **dependency service nodes**, a set of $2f+1$ Paxos acceptors, and a set of at least $f+1$ replicas. These nodes have the following responsibilities.

- The dependency service nodes, collectively called the **dependency service**, compute dependencies and maintain the dependency invariant (Invariant 2).
- The proposers and acceptors implement one instance of Paxos for every vertex and maintain the consensus invariant (Invariant 1).
- The replicas construct and execute conflict graphs and send the results of executing commands back to the clients.

More concretely, Simple BPaxos executes as follows. The numbers here correspond to the numbered arrows in Figure 5.4.

- **(1)** When a client wants to propose a state machine command x, it sends x to *any* of the proposers. Note that with MultiPaxos, only one proposer is elected leader, but in Simple BPaxos, every proposer is a leader.
- **(2) and (3)** When a proposer p_i receives a command x, from a client, it places x in a vertex with globally unique vertex id $v_x = (p_i, m)$ where m is a monotonically increasing integer local to p_i. For example, proposer p_i places the first command that it receives in vertex $(p_i, 0)$, the next command in vertex $(p_i, 1)$, the next in $(p_i, 2)$, and so on. The proposer then performs a round trip of communication with the dependency service. It sends v_x and x to the dependency service, and the dependency service replies with the dependencies $\text{deps}(v_x)$. For now, we leave this process abstract. We'll explain how the dependency service computes dependencies in Section 5.2.
- **(4) and (5)** The proposer p_i then executes Phase 2 of Paxos with the acceptors, proposing that the value $(x, \text{deps}(v_x))$ be chosen in the instance of Paxos associated with vertex $v_x = (p_i, m)$. This is analogous to a MultiPaxos leader running Phase 2, proposing the command x be chosen in the instance of Paxos associated with log entry m.

  Recall from Section 2.2 that the Paxos proposer executing round 0 can safely bypass Phase 1. By design, we predetermine that the proposer p_i leads round 0 for vertices of the form (p_i, m). This is why p_i can safely bypass Phase 1 and immediately execute Phase 2.

  In the normal case, p_i gets the value $(x, \text{deps}(v_x))$ chosen in vertex v_x. It is also possible that some other proposer erroneously concluded that p_i had failed and proposed some other value in vertex v_x, but we discuss this scenario later.
- **(6)** The proposer p_i broadcasts v_x, x, and $\text{deps}(v_x)$ to all of the replicas. The replicas add vertex v_x to their conflict graph with command x and with edges to the vertices in $\text{deps}(v_x)$. The replicas execute their conflict graphs as described in Section 5.1.
- **(7)** Once a replica executes command x, it sends the result of executing command x back to the client.

Dependency Service

The dependency service consists of $2f + 1$ dependency service nodes $d_1, \ldots, d_{2f+1}$. Every dependency service node maintains an acyclic conflict graph. These conflict graphs are similar but not equal to the conflict graph that Simple BPaxos ultimately executes.

When a proposer sends a vertex v_x with command x to the dependency service, it sends v_x and x to every dependency service node. When a dependency service node d_i receives v_x

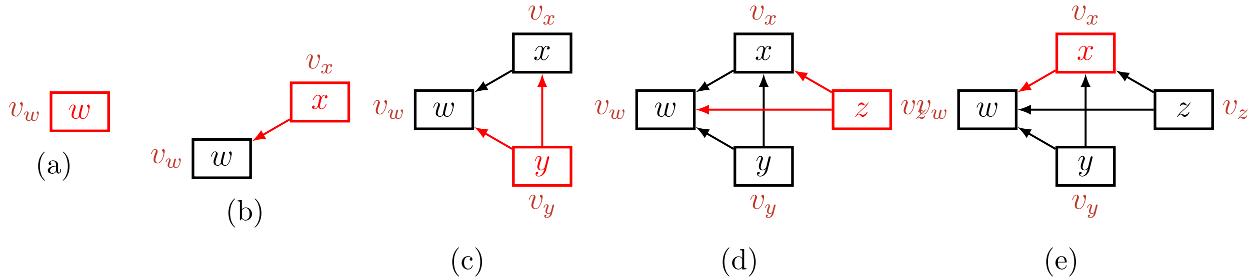


Figure 5.5: In subfigures (a) – (e), we see the execution of a dependency service node d_i. (a) d_i receives command w in vertex v_w. d_i adds this vertex to its conflict graph and because there are no other vertices, it returns the dependencies $\text{deps}(v_w) = \emptyset$. (b) d_i receives command x in vertex v_x. d_i adds this vertex to its conflict graph. x conflicts with w, so d_i adds an edge from v_x to v_w and returns the dependencies $\text{deps}(v_x) = \{v_w\}$. (c) d_i receives command y in vertex v_y. d_i adds this vertex to its conflict graph. y conflicts with w and x, so d_i adds an edge from v_y to v_w and from v_y to v_x. It returns the dependencies $\text{deps}(v_y) = \{v_w, v_x\}$. (d) d_i receives command z in vertex v_z. d_i adds this vertex to its conflict graph. z conflicts with w and x, so d_i adds an edge from v_z to v_w and from v_z to v_x. It returns the dependencies $\text{deps}(v_z) = \{v_w, v_x\}$. (e) d_i receives command x in vertex v_x. d_i's graph already contains vertex v_x, so d_i returns the dependencies $\text{deps}(v_x) = \{v_w\}$ and does not modify its graph.

and x, it performs one of the following two actions depending on whether d_i's graph already contains vertex v_x.

- If d_i's conflict graph does not contain vertex v_x, then d_i adds vertex v_x to its graph with command x. d_i adds an edge from v_x to every other vertex v_y with command y if x and y conflict. Letting $\text{out}(v_x)$ be the set of vertices to which v_x has an edge, d_i then returns $\text{out}(v_x)$ to the proposer.

- Otherwise, if d_i's conflict graph already contains vertex v_x, then d_i does not modify its conflict graph. It immediately returns $\text{out}(v_x)$ to the proposer.

An example execution of a dependency service node is given in Figure 5.5.

When a proposer receives replies from $f + 1$ dependency service nodes, it takes the union of these responses as the value of $\text{deps}(v_x)$. For example, imagine $f = 1$ and a proposer receives dependencies $\{v_w, v_y\}$ from d_1 and dependencies $\{v_w, v_z\}$ from d_2. The proposer computes $\text{deps}(v_x) = \{v_w, v_y, v_z\}$. The dependency service maintains Invariant 3.

Invariant 3. If two conflicting commands x and y in vertices v_x and v_y yield dependencies $\text{deps}(v_x)$ and $\text{deps}(v_y)$ from the dependency service, then either $v_x \in \text{deps}(v_y)$ or $v_y \in \text{deps}(v_x)$ or both.

Proof. Consider conflicting commands x and y in vertices v_x and v_y with dependencies $\text{deps}(v_x)$ and $\text{deps}(v_y)$ computed by the dependency service. $\text{deps}(v_x)$ is the union of dependencies computed by $f+1$ dependency service nodes D_x. Similarly, $\text{deps}(v_y)$ is the union of dependencies computed by $f+1$ dependency service nodes D_y. Because $f+1$ is a majority of $2f+1$, D_x and D_y necessarily intersect. That is, there is some dependency service node d_i that is in D_x and D_y. d_i either received v_x or v_y first. If it received v_x first, then it returns v_x as a dependency of v_y, so $v_x \in \text{deps}(v_y)$. If it received v_y first, then it returns v_y as a dependency of v_x, so $v_y \in \text{deps}(v_x)$. □

Note that the dependency service may process a vertex more than once, yielding different dependencies each time. For example, a proposer may sent v_x and x to the dependency service and get back dependencies $\{v_w, v_y\}$. The proposer might resend v_x and x to the dependency service and get a different set of dependencies $\{v_y, v_z\}$. Even though the dependency service may compute different dependencies for the same vertex, the dependency service still maintains Invariant 3 for every possible pair of computed dependencies.

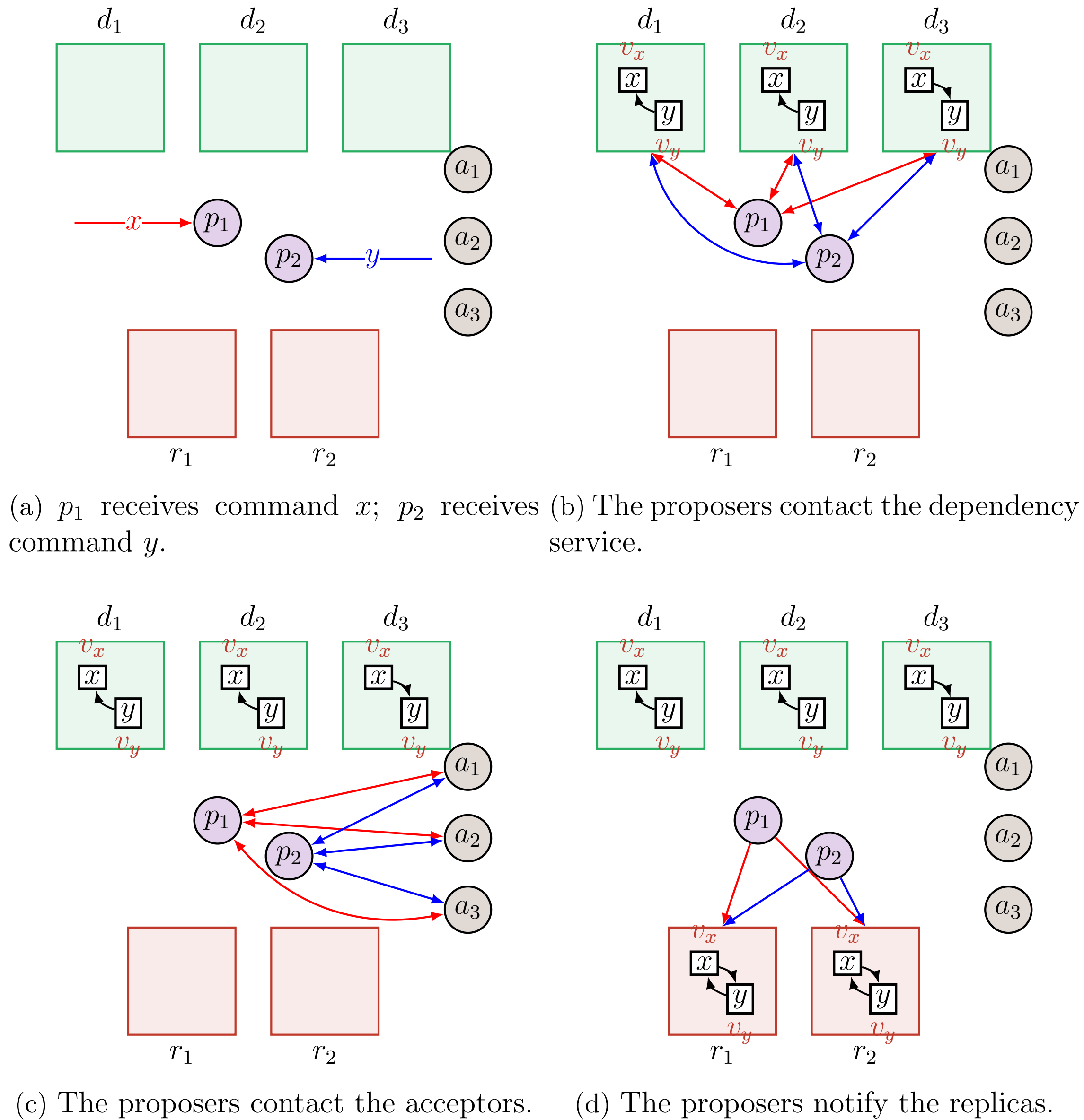
An Example

An example execution of Simple BPaxos with $f = 1$ is illustrated in Figure 5.6.

- In Figure 5.6a, proposer p_1 receives command x from a client, while proposer p_2 receives command y from a client. The commands are placed in vertices v_x and v_y respectively.
- In Figure 5.6b, p_1 sends x in v_x to the dependency service, while p_2 concurrently sends y in v_y. Dependency service nodes d_1 and d_2 receive x and then y, so they compute $\text{deps}(v_x) = \emptyset$ and $\text{deps}(v_y) = \{v_x\}$. d_3, on the other hand, receives y and then x and computes $\text{deps}(v_x) = \{v_y\}$ and $\text{deps}(v_y) = \emptyset$

  p_1 receives $\emptyset$ from d_2 and $\{v_y\}$ from d_3. Two dependency service nodes form a majority, so p_1 computes $\text{deps}(v_x) = \{v_y\} \cup \emptyset = \{v_y\}$. Similarly, p_2 receives $\{v_x\}$ from d_2 and $\emptyset$ from d_3, so p_2 computes $\text{deps}(v_y) = \{v_x\} \cup \emptyset = \{v_x\}$. Note that p_1 and p_2 also receive responses from d_1, but proposers form dependencies from the first set of $f+1$ dependency service nodes they hear from.
- In Figure 5.6c, p_1 executes Phase 2 of Paxos to get the value $(x, \{v_y\})$ chosen in vertex v_x. p_2 likewise gets the value $(y, \{v_x\})$ chosen in vertex v_y.
- In Figure 5.6d, the proposers broadcast their commands to the replicas. The replicas add v_x and v_y to their conflict graphs and execute the commands once they have received both. One or more of the replicas also sends the results of executing x and y back to the clients.

Note that the replicas' conflict graphs contain a cycle. This is because the dependency service nodes do not receive every command in the same order. In Figure 5.6, dependency service nodes d_2 and d_3 receive x and y in opposite orders, leading to the two commands



(a) p_1 receives command x; p_2 receives command y.

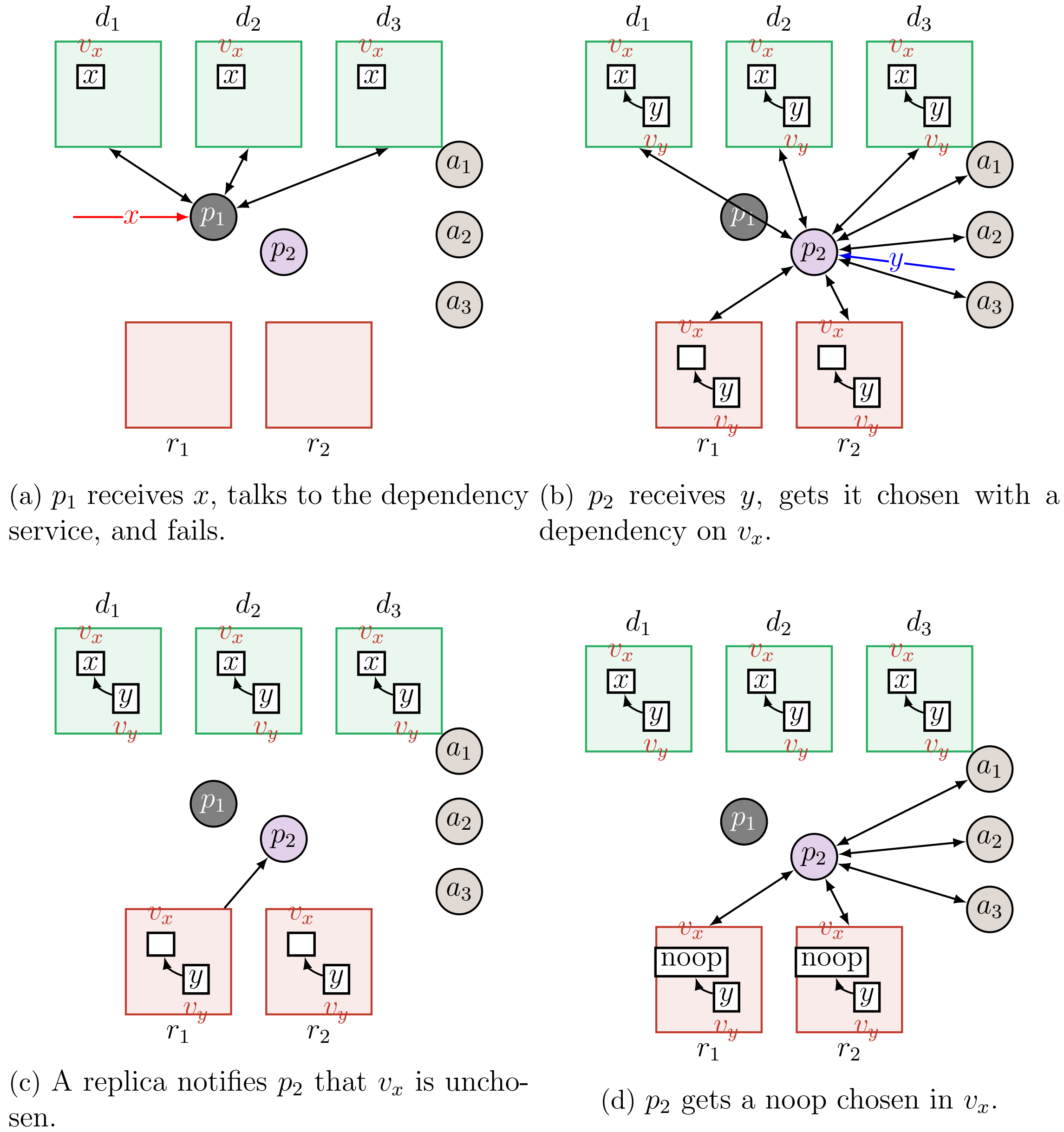
(b) The proposers contact the dependency service.

(c) The proposers contact the acceptors.

(d) The proposers notify the replicas.

Figure 5.6: An example execution of Simple BPaxos ($f = 1$).

depending on each other. It is tempting to enforce that every dependency service node receive every command in exactly the same order, but unfortunately, this would be tantamount to solving consensus [12].



(a) p_1 receives x, talks to the dependency service, and fails.

(b) p_2 receives y, gets it chosen with a dependency on v_x.

(c) A replica notifies p_2 that v_x is unchosen.

(d) p_2 gets a noop chosen in v_x.

Figure 5.7: An example execution of Simple BPaxos recovery ($f = 1$).

Recovery

Imagine a proposer receives a command x from a client, places the command x in vertex v_x, sends v_x and x to the dependency service, and then crashes. Because a command and a set of dependencies have not been chosen in vertex v_x yet, we call v_x unchosen. It is possible that a command y chosen in vertex v_y depends on an unchosen vertex v_x. If vertex v_x remains forever

unchosen, then the command y will never be executed. To avoid this liveness violation, if any replica notices that vertex v_x has been unchosen for some time, it notifies a proposer. The proposer then executes Phase 1 and Phase 2 of Paxos with the acceptors to get a **noop** chosen in vertex v_x without any dependencies. noop is a distinguished command that does not affect the state machine and does not conflict with any other command. An example of this execution is given in Figure 5.7.

- In Figure 5.7a, proposer p_1 receives command x from a client. It places x in vertex v_x and sends v_x and x to the dependency service. Shortly after, it fails.
- In Figure 5.7b, proposer p_2 receives command y from a client. It places y in v_y and contacts the dependency service. The dependency service nodes have already received x in v_x, so they compute $\text{deps}(v_y) = \{v_x\}$. p_2 then gets y chosen in vertex v_y with a dependency on v_x and broadcasts it to the replicas.
- In Figure 5.7c, the replicas cannot execute vertex v_y because it depends on the unchosen vertex v_x. After a timeout expires, replica r_1 notifies p_2 that the vertex has been unchosen for some time.
- In Figure 5.7d, p_2 executes Phase 1 and Phase 2 of Paxos in some round $r > 0$ with the acceptors to get the command noop chosen in vertex v_x without any dependencies. p_2 notifies the replicas, and the replicas place the noop in vertex v_x. The replicas execute their conflict graphs in reverse topological order. They execute the noop first (which has no effect) and then execute y.

  Note that p_2 must execute both phases of Paxos because it is not in round 0. This is necessary to ensure that no other value could have been chosen in v_x.

Note that a Simple BPaxos proposer recovers a command and proposes a noop by executing Paxos as normal. Simple BPaxos does not require an additional recovery protocol. Rather, commands and noops are proposed in the exact same way. This simplifies the protocol.

Safety

To ensure that Simple BPaxos is safe, we must ensure that it maintains the consensus invariant and the dependency invariant. Simple BPaxos maintains the consensus invariant because it implements Paxos. The dependency invariant follows immediately from Invariant 3 and the fact that noops don't conflict with any other command.

Practical Considerations

We discuss a few practical considerations that are important when implementing Simple BPaxos and the other BPaxos variants in this chapter.

Ensuring Exactly Once Semantics If a client proposes a command but does not hear back quickly enough, it re-proposes the command to make sure that the command eventually gets executed. Thus, Simple BPaxos might *receive* a command more than once, but it has to guarantee that it never *executes* the command more than once. Executing a command more than once would violate exactly once semantics.

Non-generalized protocols like MultiPaxos [87], Viewstamped Replication [53], and Raft [68] all employ the following technique to avoid executing a command more than once. First, before a client proposes a command to a replication protocol, it annotates the command with a monotonically increasing integer-valued id. Moreover, clients only send one command at a time, waiting to receive a response from one command before sending another. Second, every replica maintains a **client table**, like the one illustrated below. A client table has one entry per client. The entry for a client records the largest id of any command that the replica has executed for that client, along with the result of executing the command with that id. A replica only executes commands for a client if it has a larger id than the one recorded in the client table. If it receives a command with the same id as the one in the client table, it replies with the recorded output instead of executing the command a second time.

Client	**Id**	**Output**
10.31.14.41	2	"foo"
10.54.13.123	1	"bar"

Naively applying this same trick to Simple BPaxos (or any generalized protocol) is unsafe. For example, imagine a client issues command x with id 1. The command gets chosen and is executed by replica 1. Then, the client issues non-conflicting command y with id 2. The command gets chosen and is executed by replica 2. Because y has a larger id than x, replica 2 will never execute x.

To fix this bug, a replica must record the ids of *all* commands that it has executed for a client, along with the output corresponding to the largest of these ids. Replicas only execute commands they have not previously executed, and relay the cached output if they receive a command with the corresponding id.

Dependency Compaction Upon receiving a command x in vertex v_x, a dependency service node returns the set of all previously received vertices with commands that conflict with x. Over time, as the dependency service receives more and more commands, these dependency sets get bigger and bigger. As the dependency sets get bigger, Simple BPaxos' throughput decreases because more time is spent sending these large dependency sets, and less time is spent doing useful work.

To combat this, a dependency service node has to compact dependencies in some way. Recall that proposer p_i creates vertex ids $(p_i, 0), (p_i, 1), (p_i, 2)$, and so on. Thus, vertex ids across all the leaders form a two-dimensional array with one column for every leader index and one row for every monotonically increasing id.

Figure 5.8: An example of dependency compaction

For example, consider a dependency service node that has received commands a, b, c, d, and e in vertices $(p_0, 1)$, $(p_0, 0)$, $(p_1, 2)$, $(p_1, 0)$, and $(p_2, 1)$ as shown in Figure 5.8. *Without dependency compaction*, if the dependency service node receives a command that conflicts with commands a, b, c, d, and e, it would return the vertex ids of these five commands. In our example, the dependency service node returns only five dependencies, but in a real deployment, the node could return hundreds of thousands of dependencies.

With dependency compaction on the other hand, the dependency service node instead artificially adds more dependencies. In particular, for every proposer p_i, it computes the largest id j for which a dependency (p_i, j) exists. Then, it adds $\{(p_i, k) \,|\, k \leq j\}$ to the dependencies. In other words, it finds the largest dependency in each column and then adds all of the vertex ids below it as dependencies. In Figure 5.8, the inflated set of dependencies is highlighted in blue. Even though more dependencies have been added, the set of inflated dependencies can be represented more compactly, with a single integer for every leader (i.e., the id of the largest command for that leader). Thus, every BPaxos dependency set can be succinctly represented with n integers (for n proposers).

Scaling Following the design pattern of compartmentalization, we can scale up each component of Simple BPaxos because the protocol is fully decoupled. We could introduce proxy leaders, but because Simple BPaxos is a multi-leader protocol, there is no need. Instead, we can scale up the number of proposers directly. We can scale up the acceptors using acceptor grids or any other read-write quorum system. We can scale up the dependency service using an arbitrary quorum system, as the dependency service's correctness only relies on intersecting quorums. Note that we use a quorum system—a system in which every pair of quorums intersect—rather than a read-write quorum system. We can also scale up the number of replicas to reduce the number of replies the replicas have to send to the clients.

An interesting phenomenon happens when we fully scale up Simple BPaxos. The only component of the protocol that is not scalable is the replicas executing writes. Every replica has to execute every write no matter what. Executing writes is bottlenecked on executing the conflict graphs in reverse topological order. This process is significantly slower than executing commands in a simple log. This means that without compartmentalization, generalized

multi-leader protocols are often faster than simpler log-based protocols like MultiPaxos. But, after compartmentalization, the simpler log-based protocols are faster. Generalized protocols introduced graphs as a way to avoid bottlenecks, but after compartmentalization, the graphs become the bottlenecks.

5.3 Fast Paxos

Algorithm 3 Fast Paxos Proposer

State: a value v, initially null
State: a round i, initially -1

```
 1: upon receiving Phase2B⟨0, v'⟩ from f + maj(f + 1)
        acceptors as the proposer of round 0 with i = 0  do
 2:     if every value of v' is the same then
 3:         v' is chosen, notify the learners
 4:     else
 5:         i ← 1
 6:         proceed to line 11 viewing every Phase2B⟨0, v'⟩
            as a Phase1B⟨1, 0, v'⟩
 7: upon recovery do
 8:     i ← next largest round owned by this proposer
 9:     send Phase1A⟨i⟩ to the acceptors
10: upon receiving Phase1B⟨i, vr, vv⟩ from f + 1 acceptors do
11:     k ← the largest vr in any Phase1B⟨i, vr, vv⟩
12:     if k = −1 then
13:         v ← an arbitrary value
14:     else if k > 0 then
15:         v ← the corresponding vv in round k
16:     else if k = 0 then
17:         if there are maj(f + 1) Phase1B⟨i, 0, v'⟩
              messages for some value v' then
18:             v ← v'
19:         else
20:             v ← an arbitrary value
21:     send Phase2A⟨i, v⟩ to the acceptors
22: upon receiving Phase2B⟨i⟩ from f + 1 acceptors do
23:     v is chosen, notify the learners
```

Simple BPaxos is designed to be easy to understand, but as shown in Figure 5.6, it takes seven network delays (in the best case) between when a client proposes a command x and

Algorithm 4 Fast Paxos Acceptor

State: the largest seen round r, initially -1
State: the largest round vr voted in, initially -1
State: the value vv voted for in round vr, initially null

```
1: upon receiving value v from client do
2:     if r = −1 then
3:         r, vr, vv ← 0, 0, v
4:         send Phase2B⟨0, v⟩ to proposer of round 0
5: upon receiving Phase1A⟨i⟩ from p with i > r do
6:     r ← i
7:     send Phase1B⟨i, vr, vv⟩ to p
8: upon receiving Phase2A⟨i, v⟩ from p with i ≥ r do
9:     r, vr, vv ← i, i, v
10:    send Phase2B⟨i⟩ to p
```

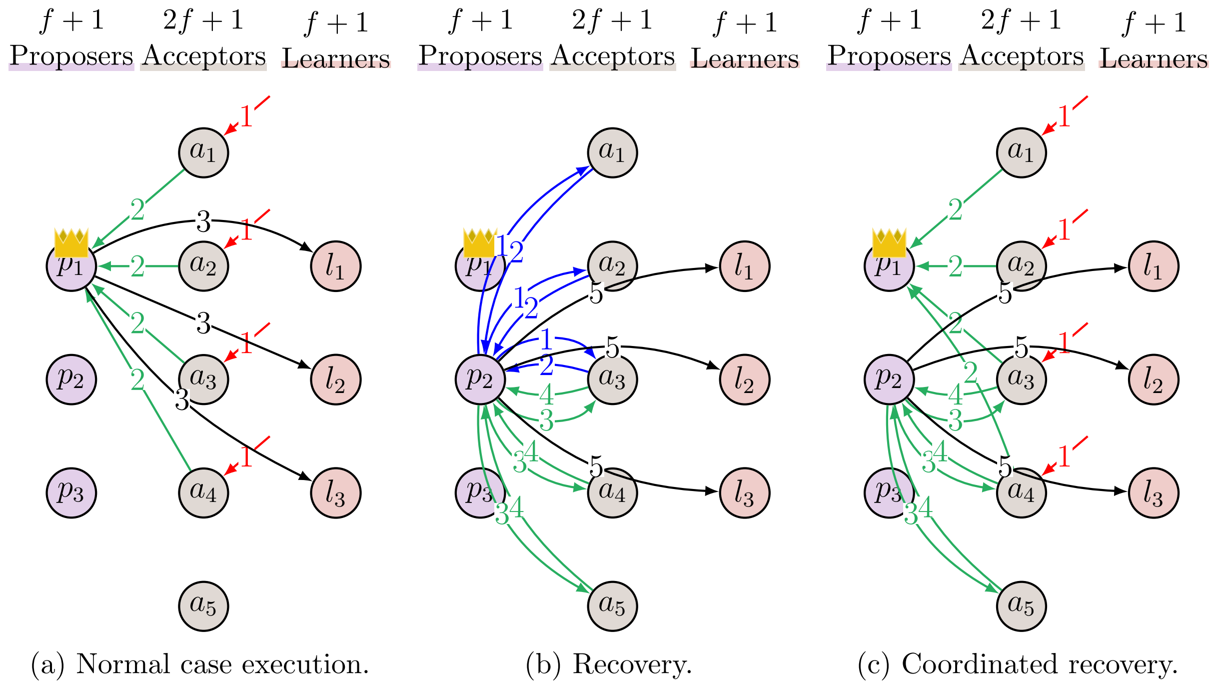
when a client receives the result of executing x. Call this duration of time the **commit time**. Generalized multi-leader protocols like EPaxos, Caesar, and Atlas all achieve a commit time of only four network delays in the best case. They do so by leveraging Fast Paxos [41].

Fast Paxos is a Paxos variant that allows clients to propose values directly to the acceptors without having to initially contact a proposer. Fast Paxos is an optimistic protocol. If all of the acceptors happen to receive the same command from the clients, then Fast Paxos has a commit time of only three network delays. This is called the fast path. However, if multiple clients concurrently propose different commands, and not all of the acceptors receive the same command, then the protocol reverts to a slow path and introduces two additional network delays to the commit time. In this section, we review a slightly simplified version of Fast Paxos.

Overview

Like Paxos, a Fast Paxos deployment consists of some number of clients, $f + 1$ proposers, and $2f + 1$ acceptors. We also include a set of $f + 1$ **learners**. These nodes are notified of the value chosen by Fast Paxos. Note that we use the term learner rather than replica because Fast Paxos is a consensus protocol and not a state machine replication protocol, so there are no state machine replicas. A Fast Paxos deployment is illustrated in Figure 5.9. Proposer and acceptor pseudocode are given in Algorithm 3 and Algorithm 4.

Like Paxos, Fast Paxos is divided into a number of integer valued rounds. The key difference is that round 0 of Fast Paxos is a special "fast round." A client can propose a value directly to an acceptor in round 0 without having to contact a proposer first. The normal case execution of Fast Paxos is illustrated in Figure 5.9a. The execution proceeds as follows:



(a) Normal case execution. (b) Recovery. (c) Coordinated recovery.

Figure 5.9: Example executions of Fast Paxos ($f = 2$). The leader of round 0 is adorned with a crown. Client requests are drawn in red. Phase 1 messages are drawn in blue. Phase 2 messages are drawn in green.

- **(1)** When a client wants to propose a value v, it sends v to all of the acceptors.
- **(2)** When an acceptor receives a value v from a client, the acceptor ignores v if it has already received a message in some round $i \geq 0$. Otherwise, it votes for v by updating its state and sending a $\textsc{Phase2B}\langle 0, v\rangle$ message to the proposer that leads round 0. This is shown in Algorithm 4 line 1 – line 4.
- **(3)** Let $\mathsf{maj}(n) = \lceil \frac{n+1}{2} \rceil$ be a majority of n. If the proposer that leads round 0 receives $\textsc{Phase2B}\langle 0, v'\rangle$ messages from $f + \mathsf{maj}(f + 1)$ acceptors for the same value v', then v' is chosen, and the proposer notifies the learners. This is shown in Algorithm 3 line 1 – line 3. We consider what happens when not every value is the same in Section 5.3.

Recovery

Note that in Paxos, a value is chosen when $f + 1$ acceptors vote for it in some round i. In round 0 of Fast Paxos, a value is chosen when $f + \mathsf{maj}(f + 1)$ acceptors vote for it. The larger number of required votes is needed to ensure the safety of recovery, which we now describe.

Let p be the proposer leading round 0. Recovery is the process by which a proposer other than p gets a value chosen. For example, if p fails, some other proposer must take over and get a value chosen. Recovery is illustrated in Figure 5.9b.

- **(1) and (2)** A recovering proposer performs Phase 1 of Paxos with at least $f + 1$ acceptors in some round $i > 0$. This is shown in Algorithm 3 line 7 – line 9 and Algorithm 4 line 5 – line 7.
- **(3) and (4)** The recovering proposer receives $\text{Phase1B}\langle i, vr, vv\rangle$ messages from $f+1$ acceptors. Call this quorum of acceptors A. The proposer computes k as the largest received vr (line 11). This is the largest round in which any acceptor in A has voted. If $k = -1$ (line 12), then none of the acceptors have voted in any round less than i, so the proposer is free to propose an arbitrary value. This is the same as in Paxos. If $k > 0$ (line 14), then the proposer must propose the value vv proposed in round k. Again, this is the same as in Paxos. vv may have been chosen in round k, so the proposer is forced to propose it as well. If $k = 0$ (line 16), then there are two cases to consider.

  First, if $\mathsf{maj}(f + 1)$ of the acceptors in A have all voted for some value v' in round 0, then it's possible that v' was chosen in round 0 (line 17). Specifically, if all f of the acceptors not in A voted for v' in round 0, then along with the $\mathsf{maj}(f+1)$ of acceptors in A who also voted for v' in round 0, there is a quorum of $f+\mathsf{maj}(f+1)$ acceptors who voted for v' in round 0. In this case, the proposer must propose v' as well since it might have been chosen. Second, if there does not exist $\mathsf{maj}(f + 1)$ votes for any value v', then the proposer concludes that no value was chosen or every will be chosen in round 0, so it is free to propose an arbitrary value (line 19). Once the recovering proposer determines which value to propose, it gets the value chosen with the acceptors using the normal Phase 2 of Paxos.

  Note that a value must receive at least $f + \mathsf{maj}(f + 1)$ votes in round 0 to be chosen. If this number were any smaller, it would be possible for a recovering proposer to find two distinct values v' and v'' that *both* could have been chosen in round 0. In this case, the proposer cannot make progress. It cannot propose v' because v'' might have been chosen, and it cannot propose v'' because v' might have been chosen

  More concretely, imagine an Fast Paxos deployment with $f = 2$ and five acceptors a_1, a_2, ..., a_5. Further imagine that a value is considered chosen after receiving votes from only 3 (i.e. $f + 1$) acceptors rather than the correct number of 4 (i.e. $f + \mathsf{maj}(f + 1)$). Consider a proposer executing Phase 1 in round 1. It contacts a_3, a_4, and a_5. a_3 voted for value x in round 0; a_4 voted for value y in round 0; and a_5 didn't vote in round 0. What value should the proposer propose in Phase 2? Well, x was maybe chosen in round 0 (if a_1 and a_2 both voted for x in round 0), so the proposer has to propose x. However, y was also maybe chosen in round 0 (if a_1 and a_2 both voted for y in round 0), so the proposer also has to propose y. The proposer can only propose one value,

so the protocol gets stuck. By requiring $f + \mathsf{maj}(f+1)$ votes rather than $f+1$ votes, we eliminate these situations. It's not possible for two values to both potentially have received $f + \mathsf{maj}(f+1)$ votes. There isn't enough acceptors for this to be possible.

- **(5)** The proposer notifies the learners of the chosen value.

Coordinated Recovery

Finally, we consider what happens when the proposer of round 0 receives $f + \mathsf{maj}(f+1)$ Phase1B messages from the acceptors, but without all of them containing the same value v'. Naively, the proposer could simply perform a recovery, executing both phases of Paxos is some round $r > 0$. However, if we assign rounds to proposers in such a way that the proposer of round 0 is also the proposer of round 1, then we can take advantage of an optimization called **coordinated recovery**. This is illustrated in Figure 5.9c and proceeds as follows:

- **(1)** Multiple clients send distinct commands directly to the acceptors.
- **(2)** The acceptors receive and vote for the commands and send Phase2B messages to the leader of round 0. However, not every acceptor receives the same value first, so not all the acceptors vote for the same value.
- **(3) and (4)** The proposer receives Phase2B messages from $f + \mathsf{maj}(f+1)$ acceptors, but the acceptors have not all voted for the same value. At this point, the proposer could naively perform a recovery in round 1 by executing Phase 1 and then Phase 2 of Paxos. But, executing Phase 1 in round 1 is redundant, since the Phase2B messages in round 0 contain exactly the same information as the Phase1B messages in round 1. Specifically, the proposer can view every Phase2B$\langle 0, v' \rangle$ message as a proxy for a Phase1B$\langle 1, 0, v' \rangle$ message. Thus, the proposer instead jumps immediately to Phase 2 in round 1 to get a value chosen (line 4 – line 6).
- **(5)** Finally, the proposer notifies the learners of the chosen value.

5.4 Fast BPaxos

In this section, we present a purely pedagogical protocol called **Fast BPaxos**. Fast BPaxos achieves a commit time of four network delays (compared to Simple BPaxos' seven), but Fast BPaxos is unsafe. It does not properly implement state machine replication. To understand why more complex protocols like EPaxos, Caesar, and Atlas work the way they do, it helps to understand why simpler protocols like Fast BPaxos don't work in the first place. Understanding why Fast BPaxos is unsafe leads to fundamental insights into these other protocols.

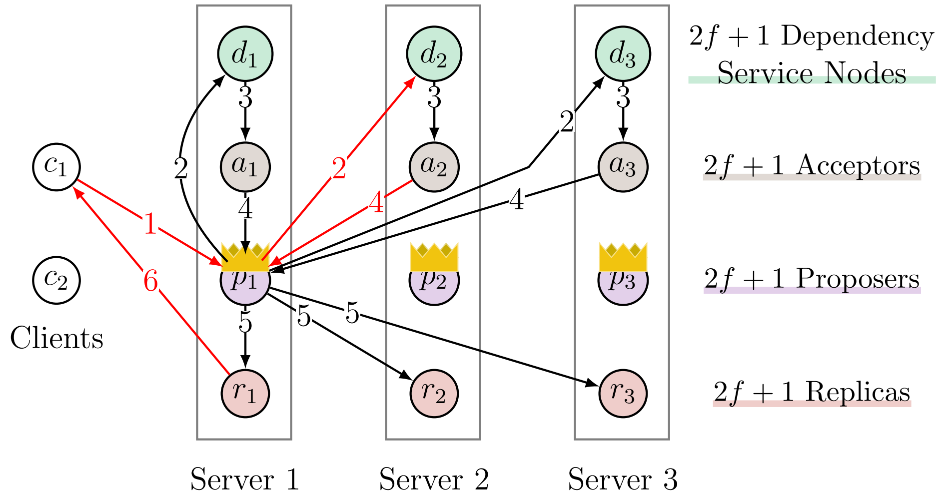


Figure 5.10: An example execution of Fast BPaxos ($f = 1$). The four network delays are drawn in red.

The Protocol

Fast BPaxos is largely identical to Simple BPaxos with one key observation. Rather than implementing Paxos, Fast BPaxos implements Fast Paxos. This allows the protocol to reduce the commit time by overlapping communication with the dependency service (to compute dependencies) and communication with the acceptors (to implement consensus).

As shown in Figure 5.10, a Fast BPaxos deployment consists of $2f+1$ dependency service nodes, $2f+1$ Fast Paxos acceptors, $2f+1$ Fast Paxos proposers, and $2f+1$ replicas. These logical nodes are co-located on a set of $2f+1$ servers, where every physical server executes one dependency service node, one acceptor, one proposer, and one replica. For example, in Figure 5.10, server 2 executes d_2, a_2, p_2, and r_2. As illustrated in Figure 5.10, the protocol executes as follows:

- **(1)** When a client wants to propose a state machine command x, it sends x to *any* of the proposers.
- **(2)** When a proposer p_i receives a command x, from a client, it places x in a vertex with globally unique vertex id $v_x = (p_i, m)$ where m is a monotonically increasing integer local to p_i. p_i then sends v_x and x to all of the dependency service nodes. Note that we predetermine that proposer p_i is the leader of round 0 and 1 of the Fast Paxos instance associated with vertex $v_x = (p_i, m)$.
- **(3)** When a dependency service node d_j receives a command x in vertex v_x, it computes a set of dependencies deps(v_x) in the exact same way as in Simple BPaxos (i.e. d_j maintains an acyclic conflict graph). Unlike Simple BPaxos however, d_j does not send

deps(v_x) back to the proposer. Instead, it proposes to the co-located Fast Paxos acceptor a_j that the value $(x, \text{deps}(v_x))$ be chosen in the instance of Fast Paxos associated with vertex v_x in round 0.

- **(4)** Fast BPaxos acceptors are unmodified Fast Paxos acceptors. In the normal case, when an acceptor a_j receives value $(x, \text{deps}(v_x))$ in vertex $v_x = (p_i, m)$, it votes for the value and sends the vote to p_i.
- **(5)** Fast BPaxos proposers are unmodified Fast Paxos proposers. In the normal case, p_i receives $f + \mathsf{maj}(f+1)$ votes for value $(x, \text{deps}(v_x))$ in vertex v_x, so $(x, \text{deps}(v_x))$ is chosen. The proposer broadcasts the command and dependencies to the replicas. If p_i receives $f + \mathsf{maj}(f+1)$ votes, but they are not all for the same value, the proposer executes coordinate recovery (see Algorithm 3 line 4 – line 6).
- **(6)** Fast BPaxos replicas are identical to Simple BPaxos replicas. Replicas maintain and execute conflict graphs, returning the results of executing commands to the clients.

Note that Figure 5.10 illustrates six steps of execution, but the commit time is only four network delays (drawn in red). Communication between co-located components (e.g., between d_1 and a_1 and between p_1 and r_1) does not involve the network and therefore does not contribute to the commit time.

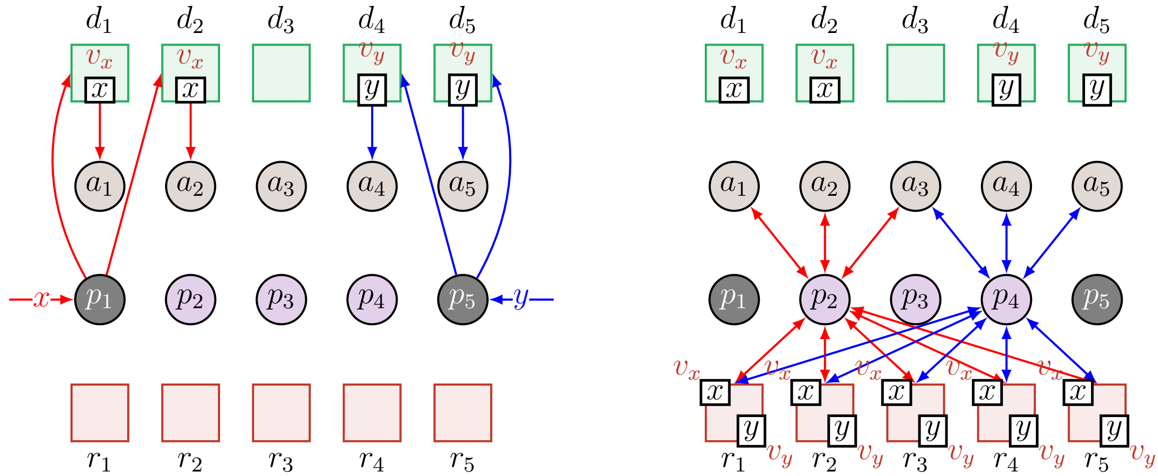
Recovery

As with Simple BPaxos, it is possible that a command y chosen in vertex v_y depends on an unchosen vertex v_x that must be recovered for execution to proceed. Fast BPaxos performs recovery in the same way as Simple BPaxos. If a replica detects that a vertex v_x has been unchosen for some time, it notifies a proposer. The proposer then executes a Fast Paxos recovery to get a noop chosen in vertex v_x with no dependencies.

Lack of Safety

We now demonstrate why Fast BPaxos is unsafe. Consider the execution of Fast BPaxos ($f = 2$) illustrated in Figure 5.11.

- In Figure 5.11a, proposer p_1 receives command x from a client. It places x in vertex v_x and sends v_x and x to the dependency service. d_1 and d_2 receive the message. They place x in their conflict graphs without any dependencies, and send the value $(x, \emptyset)$ to their co-located acceptors. a_1 and a_2 vote for $(x, \emptyset)$ in vertex v_x, but p_1 crashes before it receives any of these votes. The messages sent to d_3, d_4, and d_5 are dropped by the network.
- Similarly in Figure 5.11a, proposer p_5 receives a conflicting command y, p_5 sends v_y and y to d_4 and d_5, d_4 and d_5 propose $(y, \emptyset)$ to a_4 and a_5, a_4 and a_5 vote for the proposed values, and p_5 fails.



(a) p_1 receives x, talks to the dependency service, and fails. p_2 receives y, talks to the dependency service, and fails.

(b) p_2 recovers v_x with command x and no dependencies. p_4 recovers v_y with command y and no dependencies.

Figure 5.11: A Fast BPaxos bug ($f = 2$). Conflicting commands x and y are executed in different orders by different replicas.

- In Figure 5.11b, p_2 recovers vertex v_x. To recover v_x, p_2 executes Phase 1 of Fast Paxos with acceptors a_1, a_2, and a_3. p_2 observes that a_1 and a_2 both voted for the value $(x, \emptyset)$ in round 0. Therefore, p_2 concludes that $(x, \emptyset)$ may have been chosen in round 0, so it proceeds to Phase 2 and gets the value $(x, \emptyset)$ chosen in vertex v_x (Algorithm 3 line 17). p_2 cannot propose any other value (e.g., a noop) because $(x, \emptyset)$ may have already been chosen. From our omniscient view of the execution, we know that $(x, \emptyset)$ was never chosen, but from p_2's myopic view, it cannot make this determination and so is forced to propose $(x, \emptyset)$. This is a **critical point** in the execution, which we will discuss further in a moment.

- In Figure 5.11b, proposer p_4 recovers vertex v_y in much the same way as p_2 recovers v_x. p_4 observes that a_4 and a_5 voted for $(y, \emptyset)$ in round 0, so it is forced to get the value $(y, \emptyset)$ chosen.

- Finally in Figure 5.11b, proposers p_2 and p_4 broadcast $(x, \emptyset)$ and $(y, \emptyset)$ to all of the replicas. The replicas place x and y in their conflict graphs without edges between them. This violates the dependency invariant. x and y conflict, so there must be an edge between them. Without an edge, the replicas can execute x and y in different orders, causing their states to diverge.

What went wrong? When p_2 was recovering v_x, Fast Paxos forced it to choose $(x, \emptyset)$. However, the dependencies $\text{deps}(v_x) = \emptyset$ were *not* computed by a majority of the dependency

service nodes. They were computed only by d_1 and d_2. This is what allowed conflicting commands x and y to be chosen without a dependency on each other.

This example illustrates a **fundamental tension** between preserving the consensus invariant (Invariant 1) and preserving the dependency invariant (Invariant 2). Maintaining the consensus invariant in isolation is easy (e.g., use Paxos), and maintaining the dependency invariant in isolation is also easy (e.g., use the dependency service). But, maintaining both invariants simultaneously is tricky.

When performing a recovery, like the one in our example above, a proposer is sometimes forced to propose a particular value (e.g., $(x, \emptyset)$) in order to properly implement consensus and simultaneously forced *not* to propose the value in order to correctly compute dependencies. Resolving the tension between the consensus and dependency invariants during recovery is the single most important and the single most challenging aspect of generalized multi-leader protocols like EPaxos, Caesar, and Atlas. All of these protocols have a similar structure and execution on the normal path. They all compute dependencies from a majority of servers, and they all execute Fast Paxos variants to get these dependencies chosen. If you understand the normal case execution of one of these protocols, it is not difficult to understand the others. The key feature that distinguishes the protocols is how they resolve the fundamental tension between implementing consensus and computing dependencies.

These protocols all take different approaches to resolving the tension. In the next two sections, we broadly categorize the approaches into two main techniques: *tension avoidance* and *tension resolution*. Tension avoidance involves cleverly manipulating quorum sizes to avoid the tension entirely. This approach is used by Basic EPaxos [60] and Atlas [21]. The second technique, tension resolution, is significantly more complicated and involves detecting and resolving the tension through various means.

5.5 Tension Avoidance

In this section, we explain how to implement a generalized multi-leader state machine replication protocol using **tension avoidance**. The key idea behind tension avoidance is to avoid the tension between the consensus and dependency invariants entirely. By manipulating quorum sizes in clever ways, we can ensure that whenever a proposer is forced to propose a set of dependencies $\text{deps}(v_x)$, this set of dependencies is guaranteed to satisfy the dependency invariant.

We first introduce Unanimous BPaxos, a simple protocol that implements tension avoidance. We then explain how Basic EPaxos and Atlas can be expressed as two optimized variants of Unanimous BPaxos.

Unanimous BPaxos

A Fast BPaxos deployment consists of $2f + 1$ servers. A proposer communicates with $f + 1$ acceptors in Phase 1 called a **Phase 1 quorum**, $f + \mathsf{maj}(f + 1)$ acceptors in Phase 2 of

round 0 called a **fast Phase 2 quorum**, and $f + 1$ acceptors in Phase 2 of rounds greater than 0 called a **classic Phase 2 quorum**. If we adjust the sizes of these quorums, we can avoid the tension between implementing consensus and computing dependencies. In [30], Howard et. al prove that Fast Paxos is safe so long as the following two conditions are met.

1. Every Phase 1 quorum and every classic Phase 2 quorum intersect. That is, for every Phase 1 quorum Q and for every classic Phase 2 quorum Q', $Q \cap Q' \neq \emptyset$.

2. Every Phase 1 quorum and every pair of fast Phase 2 quorums intersect. That is, for every Phase 1 quorum Q and for every pair of fast Phase 2 quorum Q', Q'', $Q \cap Q' \cap Q'' \neq \emptyset$.

Unanimous BPaxos takes advantage of this result and increases the size of fast Phase 2 quorums. Specifically, Unanimous BPaxos is identical to Fast BPaxos except with fast Phase 2 quorums of size $2f + 1$. Unanimous BPaxos proposer pseudocode is given in Algorithm 5. It is identical to the pseudocode of a Fast Paxos proposer (Algorithm 3) except for a couple small changes highlighted in red.

Unlike Fast BPaxos, Unanimous BPaxos is safe. The critical change is on line 18. With fast Phase 2 quorums of size $2f + 1$ (line 1), a recovering proposer knows that a value v' may have been chosen in round 0 only if all $f + 1$ acceptors that it communicates with in Phase 1 voted for v' in round 0. If even a single acceptor did not vote for v' in round 0, then v' could not have received a unanimous vote and therefore was not chosen in round 0.

With Fast BPaxos, a proposer executing line 17 of Algorithm 3 is forced to propose a value $(x, \text{deps}(v_x))$ if $\mathsf{maj}(f + 1)$ acceptors voted for it in round 0, but the dependencies $\text{deps}(v_x)$ may have only been computed by $\mathsf{maj}(f + 1)$ dependency service nodes, violating the dependency invariant. This is exactly what happened in Figure 5.11. Unanimous BPaxos avoids the tension entirely because a proposer is only forced to propose a value $(x, \text{deps}(v_x))$ if $f+1$ acceptors voted for it in round 0. Now, we are guaranteed that $\text{deps}(v_x)$ was computed by a majority (i.e. $f + 1$) of the dependency service nodes. Thus, Unanimous BPaxos safely maintains the consensus and dependency service invariants.

The obvious disadvantage of Unanimous BPaxos is the protocol's large quorum sizes. In order to get a command chosen, a proposer has to perform a round trip of communication with *every* acceptor. This not only slows down the protocol in the normal case, it also decreases the protocol's ability to remain live in the face of faults. If even a single acceptor fails, the protocol grinds to a halt. This problem can be partially fixed by using more flexible quorums (like what Atlas [21] does) or by using a tension resolving protocol (see Section 5.5).

We now present two independent optimizations that improve the performance of Unanimous BPaxos. These optimizations were introduced in EPaxos [62] and Atlas [21].

Basic EPaxos Optimization

Unanimous BPaxos has a lower commit time than Simple BPaxos (4 network delays instead of 7), but has larger fast Phase 2 quorums ($2f + 1$ acceptors instead of $f + 1$). We now

Algorithm 5 Unanimous BPaxos Proposer. Changes to Algorithm 3 are highlighted in red.

State: a value v, initially null
State: a round i, initially -1

1: **upon** receiving $\text{Phase2B}\langle 0, v' \rangle$ from all $2f+1$ acceptors as the proposer of round 0 with $i = 0$ **do**
2: **if** every value of v' is the same **then**
3: v' is chosen, notify the learners
4: **else**
5: $i \leftarrow 1$
6: $v \leftarrow$ an arbitrary value satisfying the dependency invariant
7: send $\text{Phase2A}\langle i, v \rangle$ to the acceptors
8: **upon** recovery **do**
9: $i \leftarrow$ next largest round owned by this proposer
10: send $\text{Phase1A}\langle i \rangle$ to the acceptors
11: **upon** receiving $\text{Phase1B}\langle i, vr, vv \rangle$ from $f+1$ acceptors **do**
12: $k \leftarrow$ the largest vr in any $\text{Phase1B}\langle i, vr, vv \rangle$
13: **if** $k = -1$ **then**
14: $v \leftarrow$ an arbitrary value satisfying the dependency invariant
15: **else if** $k > 0$ **then**
16: $v \leftarrow$ the corresponding vv in round k
17: **else if** $k = 0$ **then**
18: **if** all $f+1$ messages are of the form $\text{Phase1B}\langle i, 0, v' \rangle$ for some value v' **then**
19: $v \leftarrow v'$
20: **else**
21: $v \leftarrow$ an arbitrary value satisfying the dependency invariant
22: send $\text{Phase2A}\langle i, v \rangle$ to the acceptors
23: **upon** receiving $\text{Phase2B}\langle i \rangle$ from $f+1$ acceptors **do**
24: v is chosen, notify the learners

discuss an optimization, used by Basic EPaxos [62], to reduce the fast Phase 2 quorum size to $2f$.

An execution of Unanimous BPaxos with the Basic EPaxos optimization is shown in Figure 5.12. We walk through the execution, highlighting the optimization's key changes. We assume $f > 1$ for now. Later, we discuss the case when $f = 1$.

- **(1)** When a client wants to propose a state machine command x, it sends x to *any* of

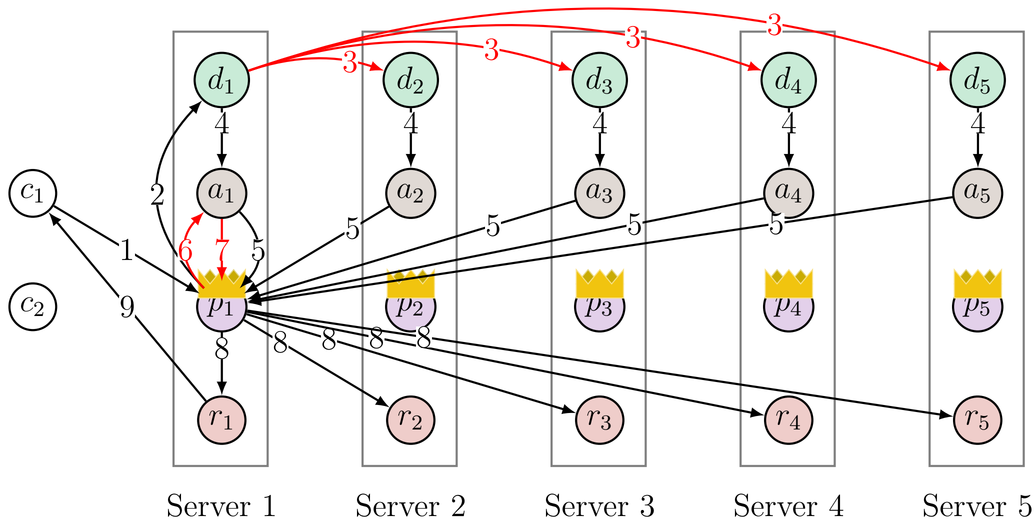


Figure 5.12: An example execution of Unanimous BPaxos ($f = 2$) with the Basic EPaxos optimization. The messages introduced by the optimization are drawn in red.

the proposers.

- **(2)** When a proposer p_i receives a command x, from a client, it places x in a vertex with globally unique vertex id $v_x = (p_i, m)$. **Change:** p_i then sends v_x and x to the co-located dependency service node d_i. It does not yet communicate with the other dependency service nodes.

- **(3)** **Change:** When d_i receives v_x and x, it computes the dependencies $\text{deps}(v_x)_i$ as usual using its acyclic conflict graph. d_i then sends v_x, x, and $\text{deps}(v_x)_i$ to all the other dependency service nodes.

- **(4)** When a dependency service node d_j receives v_x, x, and $\text{deps}(v_x)_i$, it computes the dependencies $\text{deps}(v_x)_j$ as usual using its acyclic conflict graph. **Change:** Then, d_j proposes to its co-located acceptor a_j that the value $(x, \text{deps}(v_x)_i \cup \text{deps}(v_x)_j)$ be chosen in vertex v_x in round 0. d_j combines the dependencies that it computed with the dependencies computed by d_i.

- **(5)** The acceptors are unchanged. In the normal case, when an acceptor a_j receives value $(x, \text{deps}(v_x))$ in vertex $v_x = (p_i, m)$, it votes for the value and sends the vote to p_i.

- **(6) and (7)** **Change:** In Unanimous BPaxos, a value $v = (x, \text{deps}(v_x))$ is considered chosen in round 0 if all $2f + 1$ acceptors vote for v in round 0. With the Basic EPaxos optimization, we only require $2f$ votes, and the act of choosing a value in round 0 is

made more explicit. Specifically, if p_i receives $2f$ votes for value $v = (x, \text{deps}(v_x))$ in round 0, including a vote from a_i, then it sends v to the co-located acceptor a_i. When a_i receives v and is still in round 0 (i.e. $r = 0$ on Algorithm 4 line 0), then it records v as chosen and responds to p_i. The value v is considered chosen precisely when it is recorded by the acceptor. In the future a_i responds to every PHASE1A and PHASE2A message with a notification that v is chosen. If a_i receives v but is already in a round larger than 0 (i.e. $r > 0$), then it ignores v and sends a negative acknowledgement back to p_i. Note that these messages are all performed locally on the server hosting p_i and do not incur any network delay.

- **(8)** In the normal case, p_i learns that v was successfully chosen by a_i and it broadcasts v to all the acceptors. If p_i receives a negative acknowledgement, it performs coordinated recovery as in Unanimous BPaxos.
- **(9)** The replicas are unchanged. They maintain and execute conflict graphs, returning the results of executing commands to the clients.

In addition to these changes made to the normal path of execution, the Basic EPaxos optimization also introduces a key change to the recovery procedure. Specifically, we replace line 18 – line 21 in Algorithm 5 with the following procedure.

Assume that proposer p is recovering vertex $v_x = (p_j, m)$ in round $i > 0$. Either p received a message from a_j or it did not. We consider each case separately. First, assume that p does receive a message from a_j. If p receives a message indicating that some value v' has already been chosen in round 0, then p can terminate the recovery immediately. Otherwise, p receives a PHASE1B message from a_j. From this, p can conclude that a_j is in a round at least as large as i and therefore did not and will not record any value v' chosen in round 0. Because of this, p is safe to propose any value that satisfies the dependency invariant (e.g., $(\text{noop}, \emptyset)$).

Otherwise, p does not receive a message from a_j. If p receives f PHASE1B$\langle i, 0, v' \rangle$ messages for the same value $v' = (x, \text{deps}(v_x))$, then v' may have been chosen in round 0, so p must propose v' in order to maintain the consensus invariant. Note that $\text{deps}(v_x)$ also satisfies the dependency invariant despite the fact that p only received $\text{deps}(v_x)$ from f, as opposed to $f + 1$, dependency service nodes. This is because the dependency service nodes that are not co-located with d_j all propose dependencies that include the dependencies computed by d_j. Therefore, p determines that $\text{deps}(v_x)$ is the union of $f + 1$ dependencies and maintains the dependency invariant. If p does not receive f PHASE1B$\langle i, 0, v' \rangle$ for the same value v', then p concludes no value was chosen or will be chosen in round 0, so p is safe to propose any value that satisfies the dependency invariant.

Note that when $f = 1$ and $n = 3$, Phase 1 quorums, classic Phase 2 quorums, and fast Phase 2 quorums are all of size 2. This does *not* satisfy the conditions outlined by Howard et. al [30]. As a result, our protocol as stated is not safe for $f = 1$. The reason is that a recovering proposer may receive two different values in two separate PHASE1B messages from the two non-leader acceptors with values v' and v''. In this situation, the proposer

is unable to determine which value to propose. Thankfully, we can avoid this situation by having the leader send only to $2f$ acceptors rather than to all $2f+1$ acceptors.

Ignoring some minor cosmetic differences, Unanimous BPaxos with the Basic EPaxos optimization is roughly equivalent to Basic EPaxos [62].

Atlas Optimization

In the best case, also called the fast path, Unanimous BPaxos achieves a commit time of four network delays. As shown in line 2 of Algorithm 5, a proposer executes the fast path only when every single acceptor votes for the exact same set of dependencies. As we saw in Figure 5.11, if any two dependency service nodes receive conflicting commands in different orders, their computed dependencies will not be the same. If a proposer does not receive a unanimous vote, it executes coordinated recovery, adding two more network delays to the commit time.

Atlas [21] introduces the following optimization to relax the requirement of a unanimous vote and increase the probability of a proposer executing the fast path. Let $X_1, \ldots, X_{2f+1}$ be $2f+1$ sets. Let $\mathsf{popular}(X_1, \ldots, X_{2f+1}) = \{x \mid x \text{ appears in at least } f+1 \text{ of the sets}\}$.

We change line 2 as follows. When a proposer receives dependencies $\text{deps}(v_x)_1, \ldots, \text{deps}(v_x)_{2f+1}$ from the $2f+1$ acceptors, it executes the fast path with dependencies $\text{deps}(v_x) = \text{deps}(v_x)_1 \cup \cdots \cup \text{deps}(v_x)_{2f+1}$ if $\text{deps}(v_x) = \mathsf{popular}(\text{deps}(v_x)_1, \ldots, \text{deps}(v_x)_{2f+1})$. That is, the proposer takes the fast path only if every dependency v_y computed by any of the dependency service nodes was computed by a majority of the dependency service nodes.

We also simplify line 18 – line 21. If a recovering proposer receives $f+1$ sets of dependencies, it proposes their union. Otherwise, it proposes an arbitrary value. This is safe because a set of dependencies $\text{deps}(v_x)$ can be chosen in round 0, only if every dependency in $\text{deps}(v_x)$ was computed by a majority of the dependency service nodes. Thus, every such element will appear in at least one of the $f+1$ dependency sets. Thus, the recovering proposer is sure to propose a dependency set if it was previously chosen (maintaining the consensus invariant), and it also proposes the union of $f+1$ dependency sets (maintaining the dependency invariant).

Atlas [21] is roughly equivalent to Unanimous BPaxos with the Basic EPaxos optimization, the Atlas optimization, and the flexible constraints on quorum sizes outlined in [30].

5.6 Tension Resolution

The advantage of tension avoidance is that it is simple. The disadvantage is that it requires large fast Phase 2 quorums. In this section, we explain how to implement a generalized multi-leader state machine replication protocol using **tension resolution**. Tension resolution is significantly more complicated than tension avoidance, but it does not require large fast Phase 2 quorums.

Instead of avoiding the tension between the consensus and dependency invariant, tension resolution uses additional machinery to resolve it when it arrives. Consider a scenario where a proposer p is forced to propose a set of $\text{deps}(v_x)$ in round i to maintain the consensus invariant because $\text{deps}(v_x)$ may have been chosen in a previous round. Simultaneously, p is forced not to propose $\text{deps}(v_x)$ because it cannot guarantee that $\text{deps}(v_x)$ was computed by a majority of the dependency service nodes. This is the moment of tension that tension avoidance avoids. Tension resolution, on the other hand, allows this to happen. When it does, the proposer p leverages additional machinery built into the protocol to determine either that (a) $\text{deps}(v_x)$ was not chosen or (b) $\text{deps}(v_x)$ was computed by a majority of dependency service nodes.

We introduce Majority Commit BPaxos, a protocol that implements tension resolution. We then discuss the protocol's relationship with EPaxos [62] and Caesar [21].

Pruned Dependencies

Before we discuss Majority Commit BPaxos, we introduce the notion of pruned dependencies. Imagine a proposer p sends command x to the dependency service in vertex v_x, and the dependency service computes the dependencies $\text{deps}(v_x)$. Let $v_y \in \text{deps}(v_x)$ be one of v_x's dependencies. To maintain the dependency invariant, all of the protocols that we have discussed thus far would get v_x chosen with a dependency on v_y, but this is not always necessary.

Assume that that the proposer p knows that v_y has been chosen with command y and dependencies $\text{deps}(v_y)$. Further assume that $v_x \in \text{deps}(v_y)$. That is, v_y has already been chosen with a dependency on v_x. In this case, there is no need for v_x to depend on v_y. The dependency invariant asserts that if two vertices v_a and v_b are chosen with conflicting commands a and b, then either $v_a \in \text{deps}(v_b)$ or $v_b \in \text{deps}(v_a)$. Thus, in our example, if v_y has already been chosen with a dependency on v_x, then there is no need to propose v_x with a dependency on v_y. Similarly, if v_y has been chosen with noop as part of a recovery, then there is no need to propose v_x with a dependency on v_y because x and noop do not conflict.

Let $\text{deps}(v_x)$ be a set of dependencies computed by the dependency service. Let $P \subseteq \text{deps}(v_x)$ be a set of vertices v_y such that v_y has been chosen with noop or v_y has been chosen with $v_x \in \text{deps}(v_y)$. We call $\text{deps}(v_x) - P$ the **pruned dependencies** of v_x with respect to P. Majority Commit BPaxos maintains Invariant 4, the **pruned dependency invariant**. The pruned dependency invariant is a relaxation of the dependency invariant. If a protocol maintains the pruned dependency invariant, it is guaranteed to maintain the dependency invariant.

Invariant 4 (Pruned Dependency Invariant)**.** For every vertex v_x, either $(\text{noop}, \emptyset)$ is chosen in v_x or $(x, \text{deps}(v_x) - P)$ is chosen in v_x where $\text{deps}(v_x)$ are dependencies computed by the dependency service and where $\text{deps}(v_x) - P$ are the pruned dependencies of v_x with respect to some set P.

Majority Commit BPaxos

For clarity of exposition, we first introduce a version of Majority Commit BPaxos that can sometimes deadlock. Later, we modify the protocol to eliminate the possibility of deadlock.

Majority Commit BPaxos is identical to Fast BPaxos except for the following two modifications. First, every Fast Paxos acceptor maintains a conflict graph in exactly the same way as the replicas do. That is, when an acceptor learns that a vertex v_x has been chosen with command x and dependencies $\text{deps}(v_x)$), it adds v_x to its conflict graph with command x and with edges to every vertex in $\text{deps}(v_x)$. We will see momentarily that whenever a Majority Commit BPaxos proposer sends a Phase2A message to the acceptors with value $v = (x, \text{deps}(v_x) - P)$, the proposer also sends P and all of the commands and dependencies chosen in in the vertices in P. Thus, when an acceptor receives a Phase2A message, it updates its conflict graph with the values chosen in P. Second and more substantially, a proposer executes a significantly more complex recovery procedure. This is shown in Algorithm 6.

As with Fast BPaxos, if $k = -1$ (line 3), if $k > 1$ (line 6), or if $k = 0$ and there does not exist $\mathsf{maj}(f + 1)$ matching values (line 29), recovery is straightforward.

Otherwise, there does exist a $v' = (x, \text{deps}(v_x))$ voted for by at least $\mathsf{maj}(f+1)$ acceptors in round 0 (line 9). As with Fast BPaxos, v' may have been chosen in round 0, so the proposer *must* propose v' in order to maintain the consensus invariant. But $\text{deps}(v_x)$ may not be the union of dependencies computed by $f + 1$ dependency service nodes, so the proposer is simultaneously forced *not* to propose v' in order to maintain the dependency invariant. Unanimous BPaxos avoided this tension by increasing the size of fast Phase 2 quorums. Majority Commit BPaxos instead resolves the tension by performing a more sophisticated recovery procedure. In particular, the proposer does a bit of detective work to conclude either that v' was definitely not chosen in round 0 (in which case, the proposer can propose a different value) or that $\text{deps}(v_x)$ happens to be a pruned set of dependencies (in which case, proposer is safe to propose v').

On line 11 and line 12, the proposer sends v_x and x to the dependency service nodes co-located with the acceptors in A (i.e. the $f+1$ acceptors from which the proposer received Phase1B messages). The proposer then computes the union of the returned dependencies, called $\text{deps}(v_x)_A$. Note that this communication can be piggybacked on the Phase1A messages that the proposer previously sent to avoid the extra round trip of communication. Also note that $\text{deps}(v_x)$ was returned by $\mathsf{maj}(f + 1)$ nodes in A, so $\text{deps}(v_x)$ is a subset of $\text{deps}(v_x)_A$.

Next, the proposer enters a for loop in an attempt to prune $\text{deps}(v_x)_A$ until it is equal to $\text{deps}(v_x)$. That is, the proposer attempts to construct a set of vertices P such that $\text{deps}(v_x) = \text{deps}(v_x)_A - P$ is a set of pruned dependencies. For every, $v_y \in \text{deps}(v_x)_A - \text{deps}(v_x)$, the proposer first recovers v_y if it does not know if a value has been chosen in vertex v_y (line 17). After recovering v_y, assume the proposer learns that v_y is chosen with command y and dependencies $\text{deps}(v_y)$. If $y = \text{noop}$ or if $v_x \in \text{deps}(v_y)$, then the proposer can safely prune v_y from $\text{deps}(v_x)_A$, so it adds v_y to P (line 19).

Otherwise, the proposer contacts some quorum A' of acceptors (line 21). If any acceptor a_j in A' knows that vertex v_x has already been chosen, then the proposer can abort the recovery of v_x and retrieve the chosen value directly from a_j (line 23). Otherwise, the proposer concludes that no value was chosen in v_x in round 0 and is free to propose any value that maintains the dependency invariant (line 25). We will explain momentarily why the proposer is able to make such a conclusion. It is not obvious. Note that the proposer can piggyback its communication with A' on its PHASE1A messages.

Finally, if the proposer exits the for loop, then it has successfully pruned $\text{deps}(v_x)_A$ into $\text{deps}(v_x)_A - P = \text{deps}(v_x)$ and can safely propose it without violating the consensus or pruned dependency invariant (line 28). As described above, when the proposer sends a PHASE2A message with value v', it also includes the values chosen in every vertex in P.

We now return to line 25 and explain how the proposer is able to conclude that v' was not chosen in round 0. On line 25, the proposer has already concluded that v_y was not chosen with noop and that $v_x \notin \text{deps}(v_y)$. By the pruned dependency invariant, $\text{deps}(v_y) = \text{deps}(v_y)_D - P'$ is a set of pruned dependencies where $\text{deps}(v_y)_D$ is a set of dependencies computed by a set D of $f + 1$ dependency service nodes. Because $v_x \notin \text{deps}(v_y)_D - P'$, either $v_x \notin \text{deps}(v_y)_D$ or $v_x \in P'$.

v_x cannot be in P' because if v_y were chosen with dependencies $\text{deps}(v_y)_D - P'$, then some quorum of acceptors would have received P' and learned that v_x was chosen. But, when the proposer contacted the quorum A' of acceptors, none knew that v_x was chosen, and any two quorums intersect.

Thus, $v_x \notin \text{deps}(v_y)_D$. Thus, every dependency service node in D processed instance v_y before instance v_x. If not, then a dependency service node in D would have computed v_x as a dependency of v_y. However, if every dependency service node in D processed v_y before v_x, then there cannot exist a fast Phase 2 quorum of dependency service nodes that processed v_x before v_y. In this case, $v' = (x, \text{deps}(v_x))$ could not have been chosen in round 0 because it necessitates a fast Phase 2 quorum of dependency service nodes processing v_x before v_y because $v_y \notin \text{deps}(v_x)$.

Ensuring Liveness

Majority Commit BPaxos is safe, but it is not very live. There are certain failure-free situations in which Majority Commit BPaxos can permanently deadlock. The reason for this is line 17 in which a proposer defers the recovery of one vertex for the recovery of another. There exist executions of Majority Commit BPaxos with a chain of vertices $v_1, \ldots, v_m$ where the recovery of every vertex v_i depends on the recovery of vertex $v_{i+1 \bmod m}$.

We now modify Majority Commit BPaxos to prevent deadlock. First, we change the condition under which a value is considered chosen on the fast path. A proposer considers a value $v = (x, \text{deps}(v_x))$ chosen on the fast path if a fast Phase 2 quorum F of acceptors voted for v in round 0 *and* for every vertex $v_y \in \text{deps}(v_x)$, there exists a quorum $A \subseteq F$ of $f + 1$ acceptors that knew v_y was chosen at the time of voting for v. Second, when an acceptor a_i sends a PHASE2B vote in round 0 for value $v = (x, \text{deps}(v_x))$, a_i also includes

the subset of vertices in $\text{deps}(v_x)$ that a_i knows are chosen, as well as the values chosen in these vertices. Third, proposers execute Algorithm 6 but with the lines of code shown in Algorithm 7 inserted after line 10.

We now explain Algorithm 7. On line 11, the proposer computes the subset $M \subseteq A$ of acceptors that voted for v' in round 0. On line 12, the proposer determines whether there exists some instance $v_y \in \text{deps}(v_x)$ such that no acceptor in M knows that v_y is chosen. If such an v_y exists, then v' was not chosen in round 0. To see why, assume for contradiction that v' was chosen in round 0. Then, there exists some fast Phase 2 quorum F of acceptors that voted for v' in round 0, and there exists some quorum $A' \subseteq F$ of acceptors that know v_y has been chosen. However, A and A' intersect, but no acceptor in A both voted for v' in round 0 and knows that v_y was chosen. This is a contradiction. Thus, the proposer is free to propose any value satisfying the dependency invariant.

Next, it's possible that the proposer was previously recovering instance v_z with value $(z, \text{deps}(v_z))$ and executed line 17 of Algorithm 6, deferring the recovery of instance v_z until after the recovery of instance v_x. If so and if $v_z \in \text{deps}(v_x)$, then some acceptor $a_j \in M$ knows that v_z is chosen. Thus, the proposer can abort the recovery of instance v_z and retrieve the chosen value directly from a_j (line 16). Otherwise, $v_z \notin \text{deps}(v_x)$. In this case, no value was chosen in round 0 of instance v_z, so the proposer is free to propose any value satisfying the pruned dependency invariant in instance v_z. Here's why. $v_z \notin \text{deps}(v_x)$, so every dependency service node co-located with an acceptor in M processed v_x before v_z. $|M| \geq \mathsf{maj}(f+1)$, so there strictly fewer than $f + \mathsf{maj}(f+1)$ remaining dependency service nodes that could have processed v_z before v_x (see 6WrKTnQuDWbJhuW2n/elg.smrof//:sptth). If the proposer was recovering instance v_z but deferred to the recovery of instance v_x, then $v_x \notin \text{deps}(v_z)$. In order for v_z to have been chosen in round 0 with $v_x \notin \text{deps}(v_y)$, it requires that at least $f + \mathsf{maj}(f)$ dependency service nodes processed v_z before v_x, which we just concluded is impossible. Thus, v_z was not chosen in round 0.

Majority Commit BPaxos is deadlock free for the following reason. If a proposer is recovering instance v_z and defers to the recovery of instance v_x, then either the proposer will recover v_x using line 12 of Algorithm 7 or the proposer will recover v_z using line 16 or line 18 of Algorithm 7. In either case, any potential deadlock is avoided.

EPaxos and Caesar

EPaxos [62] and Caesar [6] are two generalized multi-leader protocols that implement tension resolution. EPaxos is very similar Majority Commit BPaxos with the Basic EPaxos optimization from Section 5.5 used to reduce fast Phase 2 quorum sizes by 1. Majority Commit BPaxos and EPaxos both prune dependencies and perform a recursive recovery procedure with extra machinery to avoid deadlocks. Caesar improves on EPaxos in two dimensions. First, much like Atlas, a Caesar proposer does not require that a fast Phase 2 quorum of acceptors vote for the exact same value in order to take the fast path. Second, Caesar avoids a recursive recovery procedure. Caesar accomplishes this using a combination of logical timestamps and carefully placed barriers in the protocol.

5.7 Related Work

A Family of Leaderless Generalized Consensus Algorithms In [55], Losa et al. propose a generic generalized consensus algorithm that is structured as the composition of a generic dependency-set algorithm and a generic map-agreement algorithm. The invariants of the dependency-set and map-agreement algorithm are very similar to the consensus and dependency invariants, and the example implementations of these two algorithms in [55] form an algorithm similar to Simple BPaxos. Like Simple BPaxos, the example implementations are simple but unoptimized. Our thesis builds on this body of work by introducing Fast BPaxos, Unanimous BPaxos, and Majority Commit BPaxos. We also identify the tension between the two invariants as the key distinguishing feature of most protocols and taxonimize existing protocols by how they handle the tension. The protocols we develop and their classification based on how they handle tension brings clarity to the existing multi-leader generalized protocols. The example implementation from [55] does not correspond directly to an existing protocol.

Generalized Paxos and GPaxos Generalized Paxos [42] and GPaxos [81] are generalized protocols but are not fully multi-leader. Clients send commands directly to acceptors, behaving very much like a leader. However, in the face of collisions, Generalized Paxos and GPaxos rely on a single leader to resolve the collision. This single leader becomes a bottleneck in high contention workloads and prevents scaling.

SpecPaxos, NOPaxos, CURP SpecPaxos [71] and NOPaxos [51] combine speculative execution and ideas from Fast Paxos in order to reduce commit delay as low as two network delays. CURP [69] further introduces generalization, allowing commuting commands to be executed in any order. These protocols represent yet another point in the design space of state machine replication protocols. As the commit delay decreases, the complexity of the protocols generally increases. We think this is an exciting avenue of research and hope that an improved understanding of generalized multi-leader protocols can accelerate research in this direction.

Mencius Mencius [57] is a multi-leader, non-generalized protocol in which MultiPaxos log entries are round-robin partitioned among a set of leaders. Because Mencius is not generalized, a log entry cannot be executed until *all* previous log entries have been executed. To ensure log entries are being filled in appropriately, Mencius leaders perform all-to-all communication between each other. Mencius is significantly less complex that generalized multi-leader protocols like EPaxos, Caesar, and Atlas. This demonstrates that much of the complexity of these protocols come from being generalized rather than being multi-leader, though both play a role.

Chain Replication Chain Replication [88] is a state machine replication protocol in which the set of state machine replicas are arranged in a totally ordered chain. Writes are propagated through the chain from head to tail, and reads are serviced exclusively by the tail. Chain Replication has high throughput compared to MultiPaxos because load is more evenly distributed between the replicas. This shows that the leader bottleneck can be addressed without necessarily having multiple leaders.

Scalog Scalog [20] is a replicated shared log protocol that achieves high throughput using a sophisticated form of batching. A client does not send values directly to a centralized leader for sequencing in the log. Instead, the client sends its values to one of a number of batchers. Periodically, the batchers' batches are sealed and assigned an id. This id is then sent to a state machine replication protocol, like MultiPaxos, for sequencing. Like Mencius, Scalog represents a way to avoid a leader bottleneck without needing multiple leaders.

PQR, Harmonia, and CRAQ PQR [13], Harmonia [98], and CRAQ [85] all implement optimizations so that reads (i.e. state machine commands that do not modify the state of the state machine) can be executed without contacting a leader, while writes are still processed by the leader. An interesting direction of future work could explore whether or not these read optimizations could be applied to generalized multi-leader protocols.

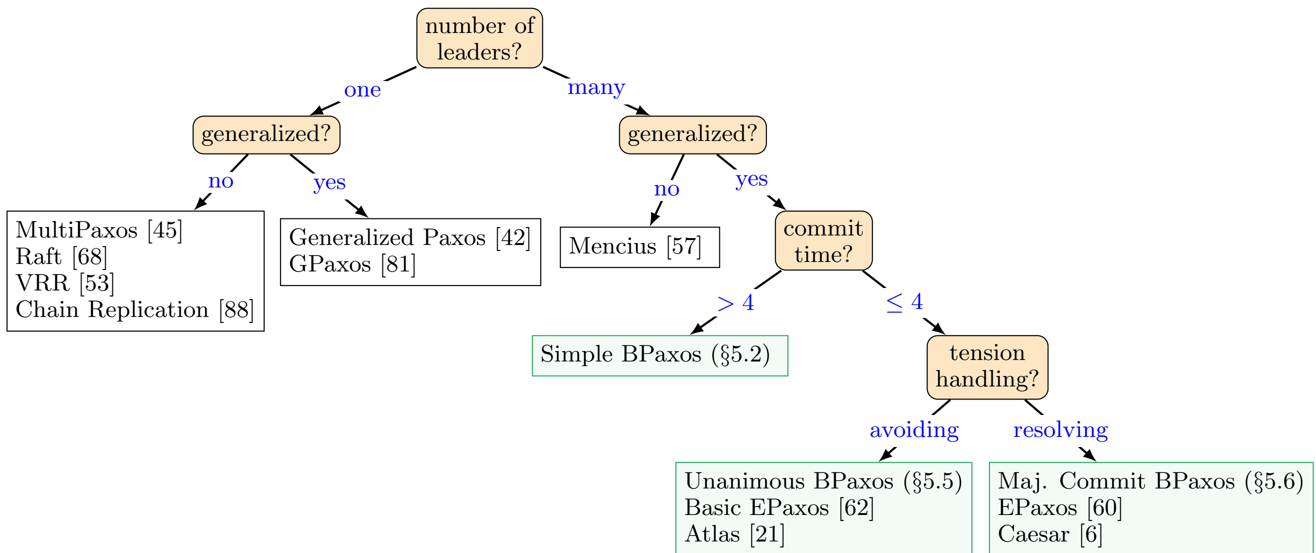


Figure 5.13: A non-exhaustive taxonomy of state machine replication protocols. The generalized multi-leader protocols that we discuss in this chapter are shaded green.

5.8 Conclusion

In this chapter, we explained, analyzed, and taxonomized generalized multi-leader state machine replication protocols. Our taxonomy of state machine replication protocols is summarized in Figure 5.13, and a summary of the generalized multi-leader protocols that we discuss in this chapter is given in Table 5.2.

Algorithm 6 Majority Commit BPaxos Proposer. Pseudocode for initiating recovery and handling PHASE2B messages is ommitted because it is identical to the pseudocode in Algorithm 3.

State: a value v, initially null
State: a round i, initially -1

1: **upon** receiving PHASE1B$\langle i, vr, vv\rangle$ from $f+1$ acceptors A **do**
2: $k \leftarrow$ the largest vr in any PHASE1B$\langle i, vr, vv\rangle$
3: **if** $k = -1$ **then**
4: $v \leftarrow$ an arbitrary value satisfying the dependency invariant
5: send PHASE2A$\langle i, v\rangle$ to the acceptors
6: **else if** $k > 0$ **then**
7: $v \leftarrow$ the corresponding vv in round k
8: send PHASE2A$\langle i, v\rangle$ to the acceptors
9: **else if** there are maj$(f+1)$ PHASE1B$\langle i, 0, v'\rangle$ messages for some value v' **then**
10: $(x, \text{deps}(v_x)) \leftarrow v'$
11: send v_x and x to the dependency service nodes co-located with the acceptors in A
12: $\text{deps}(v_x)_A \leftarrow$ the union of the dependencies returned by these dependency service nodes
13:
14: $P \leftarrow \emptyset$
15: **for** $v_y \in \text{deps}(v_x)_A - \text{deps}(v_x)$ **do**
16: **if** we don't know if v_y is chosen **then**
17: recover v_y, blocking until v_y is recovered
18: **if** v_y chosen with noop or with $v_x \in \text{deps}(v_y)$ **then**
19: $P \leftarrow P \cup \{v_y\}$
20: **else**
21: contact a quorum A' of acceptors
22: **if** an acceptor in A' knows v_x is chosen **then**
23: abort recovery; v_x has already been chosen
24: **else**
25: $v \leftarrow$ an arbitrary value satisfying the dependency invariant
26: send PHASE2A$\langle i, v\rangle$ to the acceptors
27: $v \leftarrow v'$
28: send PHASE2A$\langle i, v\rangle$ and the values chosen in P to at least $f+1$ acceptors
29: **else**
30: $v \leftarrow$ an arbitrary value satisfying the dependency invariant
31: send PHASE2A$\langle i, v\rangle$ to the acceptors

Algorithm 7 Majority Commit BPaxos proposer modification to prevent deadlock.

11: $M \leftarrow$ the set of acceptors in A that voted for v' in round 0
12: **if** $\exists v_y \in \text{deps}(v_x)$ such that no acceptor in M knows that v_y is chosen **then**
13:     send any value satisfying the dependency invariant
14: **if** the proposer was recovering v_z and deferred to the recovery of v_x **then**
15:     **if** $v_z \in \text{deps}(v_x)$ **then**
16:         abort recovery of v_z; v_z has already been chosen
17:     **else**
18:         in vertex v_z, send any value satisfying the dependency invariant

Table 5.2: A summary of generalized multi-leader state machine replication protocols.

Protocol	Safe	Commit Time	Tension Handling	Number of Nodes	Phase 1 Quorum Size	Classic Phase 2 Quorum Size	Fast Phase 2 Quorum Size
Simple BPaxos (§5.2)	yes	7	N/A	$2f+1$	$f+1$	$f+1$	N/A
Fast BPaxos (§5.4)	no	4	N/A	$2f+1$	$f+1$	$f+1$	$f+\mathsf{maj}(f+1)$
Unanimous BPaxos (§5.5)	yes	4	avoidance	$2f+1$	$f+1$	$f+1$	$2f+1$
Basic EPaxos [62]	yes	4	avoidance	$2f+1$	$f+1$	$f+1$	$2f$
Atlas [21]	yes	4	avoidance	n	$f+1$	$n-f$	$\lfloor \frac{n}{2} \rfloor + f$
Maj. Commit BPaxos (§5.6)	yes	4	resolution	$2f+1$	$f+1$	$f+1$	$f+\mathsf{maj}(f+1)$
EPaxos [60]	yes	4	resolution	$2f+1$	$f+1$	$f+1$	$f+\mathsf{maj}(f+1)-1$
Caesar [6]	yes	4	resolution	$2f+1$	$f+1$	$f+1$	$f+\mathsf{maj}(f+1)$

Chapter 6

Matchmaker Paxos

In Chapter 3, we saw how to use compartmentalization to make state machine replication protocols faster, and in Chapter 5 we saw how to use compartmentalization to make state machine replication protocols easier to understand. Now, we see how to use compartmentalization to implement new features in a state machine replication protocol. Specifically, we design a reconfiguration protocol.

Over time, machines fail, and if too many machines in a state machine replication protocol fail, the protocol grinds to a halt. Thus, state machine replication protocols have to replace failed machines with new machines as the protocol runs, a process known as reconfiguration.

Reconfiguration is an essential component of state machine replication. It is not an optimization or an afterthought. Without a reconfiguration protocol in place, a state machine replication protocol will inevitably stop working. Despite this, reconfiguration has largely been neglected by current academic literature. Researchers have invented dozens of state machine replication protocols, yet many papers either discuss reconfiguration briefly with no evaluation [62, 71, 74, 73], propose theoretically safe but inefficient reconfiguration protocols [42, 53], or do not discuss reconfiguration at all [41, 57, 58, 10, 6].

Ignoring reconfiguration has never been ideal, but we have largely been able to do so without consequence. Historically, state machine replication protocols were deployed on a fixed set of machines, and reconfiguration was used only to replace failed machines with new machines – an infrequent occurrence. This made it easy to ignore reconfiguration. Recently however, systems have become increasingly elastic, and the need for frequent reconfiguration has grown. These elastic systems do not just perform reconfigurations *reactively* when machines fail; they reconfigure *proactively*. For example, cloud databases can proactively request more resources to handle workload spikes, and orchestration tools like Kubernetes [38] are making it easier to build these types of elastic systems. Similarly, in environments with short-lived cloud instances—as with serverless computing and spot instances—and in mobile edge and Internet of Things settings, protocols must adapt to a changing set of machines much more frequently. This frequent need for reconfiguration makes it hard to ignore reconfiguration any longer.

In this chapter, we present a reconfigurable consensus protocol and a reconfigurable

state machine replication protocol: Matchmaker Paxos and Matchmaker MultiPaxos. In a nutshell, our protocols work by leveraging two key compartmentalizations.

- The first is to decouple reconfiguration from the standard processing path. Many replication protocols [47, 67, 62, 53] have machines that are responsible for both processing commands and for orchestrating reconfigurations. By contrast, Matchmaker Paxos introduces a set of distinguished matchmaker machines that are solely responsible for managing reconfigurations and operate off of the critical path. These matchmakers act as a source of truth; they always know the current configuration.

  Decoupling reconfiguration from the standard processing path has a number of advantages. For example, because the matchmakers are a separate entity that is off the critical path, we can reconfigure them without impacting normal case performance at all. Similarly, the matchmakers are often idle and experience very light load even in the worst case. Thus, the matchmakers can be deployed on very inexpensive machines. The decoupling also allows us to understand the standard processing path and the reconfiguration path in isolation and prove them correct separately.

- The second design point is to reconfigure across rounds, a technique known as *vertical reconfiguration* [49] and originally used by Vertical Paxos. With vertical reconfiguration, every round of consensus can execute using a different configuration. In other words, we decouple the configurations used across rounds. This is in contrast to the other commonly used technique of *horizontal reconfiguration* [87] in which the configurations used across log entries are decoupled.

At the beginning of every round, the Paxos leader queries the matchmakers to discover the older configurations that were used in previous rounds, and it simultaneously sends the matchmakers the configuration it intends to use in the current round. In this way, the matchmakers act as a registry for configurations. Leaders simultaneously query the past and update the present. This matchmaking phase requires a single round trip of communication and happens rarely. We also introduce a number of novel protocol optimizations to perform the matchmaking completely off the critical path to avoid degrading performance. Moreover, the protocol employs a garbage collection protocol to delete old configurations stored on the matchmakers. Our protocols have the following desirable properties.

- **Little to No Performance Degradation.** Matchmaker MultiPaxos can perform a reconfiguration without significantly degrading the throughput or latency of processing client commands. For example, we show that reconfiguration has less than a 4% effect on the median of throughput and latency measurements (Section 6.5). Note that Matchmaker MultiPaxos is not the first protocol to achieve this [54].

- **Quick Reconfiguration.** Matchmaker MultiPaxos can perform a reconfiguration quickly. Reconfiguring to a new set of machines takes one round trip of communication in the normal case (Section 6.2). Empirically, this requires only a few milliseconds

within a single data center (Section 6.5). It takes slightly longer to shut down the old machines, but empirically this takes only five milliseconds within a data center (Section 6.5).

- **Generality** Replication protocols based on classical MultiPaxos assume a totally ordered log of chosen commands and reconfigure across log entries, known as *horizontal reconfiguration*. However, many state machine replication protocols, like those of Chapter 5, do not replicate a log [42, 62, 6, 74, 97, 82]. These protocols cannot use horizontal reconfiguration. However, while none of these protocols have logs, they all have rounds and can implement vertical reconfiguration. This allows Matchmaker Paxos and Matchmaker MultiPaxos to serve as a foundation on top of which reconfiguration protocols can be built for these other non-log based protocols.
- **Theoretical Insights.** Matchmaker Paxos generalizes Vertical Paxos [49], it is the first protocol to achieve the theoretical lower bound on Fast Paxos [41] quorum sizes, and it corrects errors in DPaxos [65] (Section 6.4).
- **Proven Safe.** We describe Matchmaker Paxos and Matchmaker MultiPaxos precisely and prove that both are safe (Sections 6.1, 6.2, 6.3). Unfortunately, this is not often done for all reconfiguration protocols [60, 71, 74, 73].

6.1 Matchmaker Paxos

We now present Matchmaker Paxos. To ease understanding, we first describe a simplified version of Matchmaker Paxos that is easy to understand but is also naively inefficient. We then upgrade the protocol to the complete, efficient version by way of a number of optimizations.

Overview and Intuition

Matchmaker Paxos is largely identical to Paxos. Like Paxos, a Matchmaker Paxos deployment includes an arbitrary number of clients, a set of at least $f+1$ proposers, and some set of acceptors, as illustrated in Figure 6.1. Paxos assumes that a *single, fixed* read-write quorum system, which we'll call a **configuration** throughout the chapter, of acceptors is used for every round. The big difference between Paxos and Matchmaker Paxos is that Matchmaker Paxos allows every round to have a *different* configuration of acceptors. Round 0 may use some configuration C_0, while round 1 may use some completely different configuration C_1. This idea was first introduced by Vertical Paxos [49].

Recall from Section 2.2 that a Paxos proposer in round i executes Phase 1 in order to (1) learn of any value that may have been chosen in a round less than i and (2) prevent any new values from being chosen in any round less than i. To do so, the proposer contacts the *fixed set* of acceptors. A Matchmaker Paxos proposer must also execute Phase 1 to

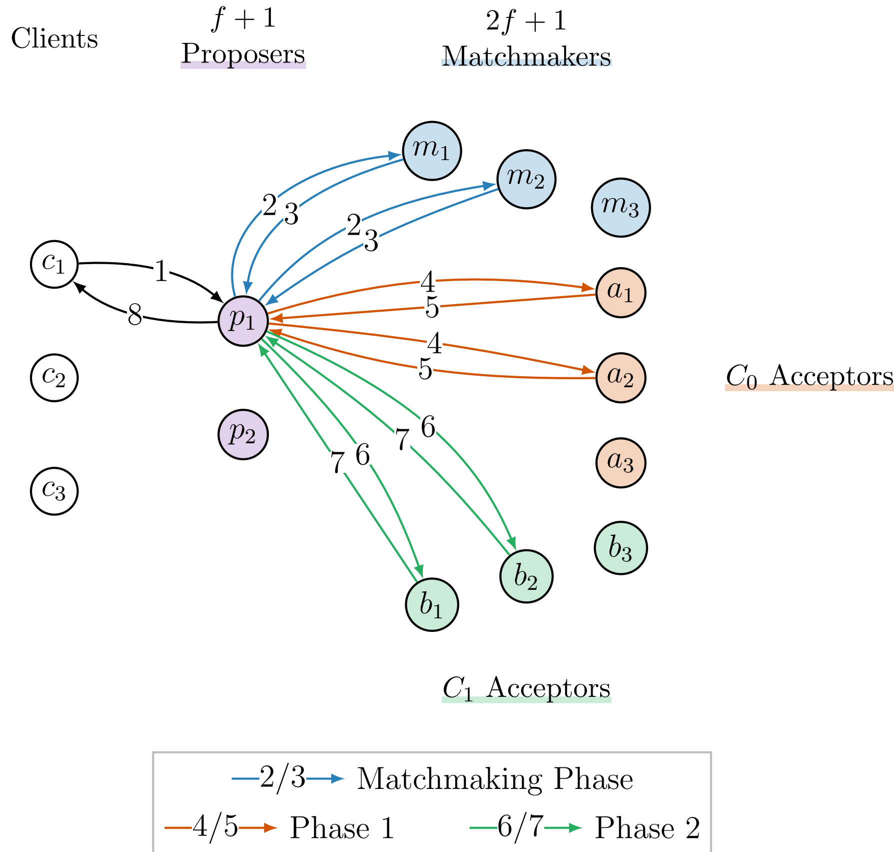


Figure 6.1: Matchmaker Paxos ($f = 1$).

establish that these two properties hold. The difference is that there is no longer a single fixed configuration of acceptors to contact. Instead, a Matchmaker Paxos proposer has to contact all of the configurations used in rounds less than i.

However, every round can use a different configuration of acceptors, so how does the proposer of round i know which acceptors to contact in Phase 1? To resolve this question, a Matchmaker Paxos deployment also includes a set of $2f + 1$ **matchmakers**. The protocol executes as follows, as illustrated in Figure 6.1.

(1) A client proposes a value x by sending it to a proposer (p_1 in Figure 6.1).

(2,3) When a proposer receives a value x, it begins executing the protocol in some round i. It selects a configuration C_i and sends C_i to the matchmakers. The matchmakers reply with the configurations used in previous rounds. We call this the **Matchmaking phase**. In Figure 6.1, the proposer executes in round 1 and selects configuration C_1 consisting of acceptors b_1, b_2, and b_3. The matchmakers reply with the configuration C_0 consisting of acceptors a_1, a_2, and a_3.

(4,5) The proposer then executes Phase 1 of Paxos with the prior configurations that it received during the Matchmaking Phase. In Figure 6.1, the proposer executes Phase 1 with configuration C_0.

(6,7) The proposer then executes Phase 2 with the configuration C_i to get the value x chosen. In Figure 6.1, the proposer executes Phase 2 with configuration C_1.

(8) Finally, the proposer informs the client that x was chosen.

At first, the extra round trip of communication with the matchmakers and the large number of configurations in Phase 1 make Matchmaker Paxos look slow. This is for ease of explanation. Later, we will eliminate these costs (Section 6.1 – Section 6.1).

Details

Every matchmaker maintains a log L of configurations indexed by round. That is, $L[i]$ stores the configuration of round i. When a proposer receives a request x from a client and begins executing round i, it first selects a configuration C_i to use in round i. It then sends a MatchA$\langle i, C_i \rangle$ message to all of the matchmakers.

When a matchmaker receives a MatchA$\langle i, C_i \rangle$ message, it checks to see if it had previously received a MatchA$\langle j, C_j \rangle$ message for some round $j \geq i$. If so, the matchmaker ignores the MatchA$\langle i, C_i \rangle$ message. Otherwise, it inserts C_i in log entry i and computes the set H_i of previous configurations in the log: $H_i = \{(j, C_j) \mid j < i, C_j \in L\}$. It then replies to the proposer with a MatchB$\langle i, H_i \rangle$ message. Matchmaker pseudocode is given in Algorithm 8. An example execution of a matchmaker is illustrated in Figure 6.2.

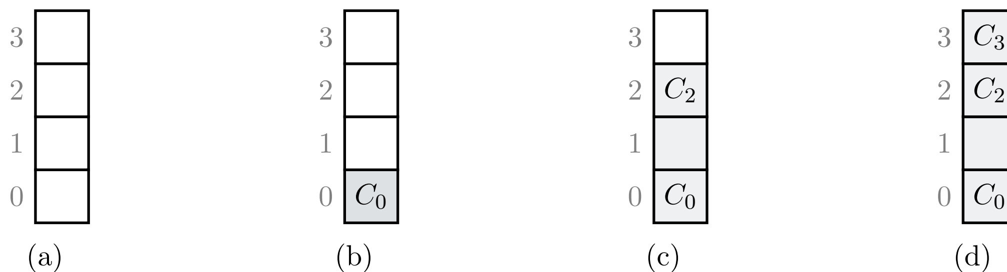


Figure 6.2: A matchmaker's log over time. (a) Initially, the matchmaker's log is empty. (b) Then, the matchmaker receives MatchA$\langle 0, C_0 \rangle$. It inserts C_0 in log entry 0 and returns MatchB$\langle 0, \emptyset \rangle$ since the log does not contain any configuration in any round less than 0. (c) The matchmaker then receives MatchA$\langle 2, C_2 \rangle$. It inserts C_2 in log entry 2 and returns MatchB$\langle 2, \{(0, C_0)\} \rangle$. (d) It then receives MatchA$\langle 3, C_3 \rangle$, inserts C_3 in log entry 3, and returns MatchB$\langle 3, \{(0, C_0), (2, C_2)\} \rangle$. At this point, if the matchmaker were to receive MatchA$\langle 1, C_1 \rangle$, it would ignore it.

Algorithm 8 Matchmaker Pseudocode

State: a log L indexed by round, initially empty
1: **upon** receiving MatchA$\langle i, C_i \rangle$ from proposer p **do**
2: **if** $\exists$ a configuration C_j in round $j \geq i$ in L **then**
3: ignore the MatchA$\langle i, C_i \rangle$ message
4: **else**
5: $H_i \leftarrow \{(j, C_j) \mid C_j \in L\}$
6: $L[i] \leftarrow C_i$
7: send MatchB$\langle i, H_i \rangle$ to p

When the proposer in round i receives MatchB$\langle i, H_i^1 \rangle$, ..., MatchB$\langle i, H_i^{f+1} \rangle$ from $f+1$ matchmakers, it computes $H_i = \cup_{j=1}^{f+1} H_i^j$. For example, with $f = 1$ and $i = 2$, if the proposer in round 2 receives MatchB$\langle 2, \{(0, C_0)\} \rangle$ and MatchB$\langle 2, \{(1, C_1)\} \rangle$, it computes $H_2 = \{(0, C_0), (1, C_1)\}$. Note that every round is statically assigned to a single proposer and that a proposer selects a single configuration for a round, so if two matchmakers return configurations for the same round, they are guaranteed to be the same.

The proposer then ends the Matchmaking phase and begins Phase 1. It sends Phase1A messages to every acceptor in every configuration in H_i and waits to receive Phase1B messages from a Phase 1 quorum from every configuration. Using the previous example, the proposer sends Phase1A messages to every acceptor in C_0 and C_1 and waits for Phase1B messages from a Phase 1 quorum of C_0 and a Phase 1 quorum of C_1. The proposer then runs Phase 2 with C_i.

Acceptor and proposer pseudocode are shown in Algorithm 9 and Algorithm 10 respectively. To keep things simple, we assume that round numbers are integers, but generalizing to an arbitrary totally ordered set is straightforward. A Matchmaker Paxos acceptor is identical to a Paxos acceptor. A Matchmaker Paxos proposer is nearly identical to a Flexible Paxos proposer with the exception of the Matchmaking phase and the configurations used in Phase 1 and Phase 2. For clarity of exposition, we omit straightforward details surrounding re-sending dropped messages and nacking ignored messages.

Algorithm 9 Acceptor Pseudocode

State: the largest seen round r, initially -1
State: the largest round vr voted in, initially -1
State: the value vv voted for in round vr, initially `null`
1: **upon** receiving Phase1A$\langle i \rangle$ from p with $i > r$ **do**
2: $r \leftarrow i$
3: send Phase1B$\langle i, vr, vv \rangle$ to p
4: **upon** receiving Phase2A$\langle i, x \rangle$ from p with $i \geq r$ **do**
5: $r, vr, vv \leftarrow i, i, x$
6: send Phase2B$\langle i \rangle$ to p

Algorithm 10 Proposer Pseudocode. Modifications to a Paxos proposer are underlined and shown in blue.

State: a value x, initially null
State: a round i, initially -1
State: the configuration C_i for round i, initially null
State: the prior configurations H_i for round i, initially null

1: **upon** receiving value y from a client **do**
2: $i \leftarrow$ next largest round owned by this proposer
3: $x \leftarrow y$
4: $C_i \leftarrow$ an arbitrary configuration
5: send $\text{MATCHA}\langle i, C_i \rangle$ to all of the matchmakers
6: **upon** receiving $\text{MATCHB}\langle i, H_i^1 \rangle, \ldots, \text{MATCHB}\langle i, H_i^{f+1} \rangle$ from $f+1$ matchmakers **do**
7: $H_i \leftarrow \bigcup_{j=1}^{f+1} H_i^j$
8: send $\text{PHASE1A}\langle i \rangle$ to every acceptor in H_i
9: **upon** receiving $\text{PHASE1B}\langle i, -, - \rangle$ from a Phase 1 quorum from every configuration in
10: H_i **do**
11: $k \leftarrow$ the largest vr in any $\text{PHASE1B}\langle i, vr, vv \rangle$
12: **if** $k \neq -1$ **then**
13: $x \leftarrow$ the corresponding vv in round k
14: send $\text{PHASE2A}\langle i, x \rangle$ to every acceptor in C_i
15: **upon** receiving $\text{PHASE2B}\langle i \rangle$ from a Phase 2 quorum **do**
16: x is chosen, inform the client

Proof of Safety

We now prove that Matchmaker Paxos is safe; i.e. every execution of Matchmaker Paxos chooses at most one value.

Proof. Our proof is based on the Paxos safety proof in [41]. We prove, for every round i, the statement $P(i)$: "if a proposer proposes a value v in round i (i.e. sends a PHASE2A message for value v in round i), then no value other than v has been or will be chosen in any round less than i." At most one value is ever proposed in a given round, so at most one value is ever chosen in a given round. Thus, $P(i)$ suffices to prove that Matchmaker Paxos is safe for the following reason. Assume for contradiction that Matchmaker Paxos chooses distinct values x and y in rounds j and i with $j < i$. Some proposer must have proposed y in round i, so $P(i)$ ensures us that no value other than y could have been chosen in round j. But, x was chosen, a contradiction.

We prove $P(i)$ by strong induction on i. $P(0)$ is vacuous because there are no rounds less than 0. For the general case $P(i)$, we assume $P(0), \ldots, P(i-1)$. We perform a case analysis on the proposer's pseudocode (Algorithm 10). Either k is -1 or it is not (line 11).

First, assume it is not. In this case, the proposer proposes x, the value proposed in round k (line 12). We perform a case analysis on round j to show that no value other than x has been or will be chosen in any round $j < i$. That is, we show $P(i)$.

Case 1: $j > k$. We show that no value has been or will be chosen in round j. Recall that at the end of the Matchmaking phase, the proposer computed the set H_i of prior configurations using responses from a set M_i of $f + 1$ matchmakers. Either H_i contains a configuration C_j in round j or it doesn't.

First, suppose it does. Then, the proposer sent Phase1A$\langle i \rangle$ messages to all of the acceptors in C_j. A Phase 1 quorum of these acceptors, say Q, all received Phase1A$\langle i \rangle$ messages and replied with Phase1B messages. Thus, every acceptor in Q set its round r to i, and in doing so, promised to never vote in any round less than i. Moreover, none of the acceptors in Q had voted in any round greater than k. So, every acceptor in Q has not voted and never will vote in round j. For a value v' to be chosen in round j, it must receive votes from some Phase 2 quorum Q' of round j acceptors. But, Q and Q' necessarily intersect, so this is impossible. Thus, no value has been or will be chosen in round j.

Now suppose that H_i does *not* contain a configuration for round j. H_i is the union of $f + 1$ MatchB messages from the $f + 1$ matchmakers in M_i. Thus, if H_i does not contain a configuration for round j, then none of the MatchB messages did either. This means that for every matchmaker $m \in M_i$, when m received MatchA$\langle i, C_i \rangle$, it did not contain a configuration for round j in its log. Moreover, by processing the MatchA$\langle i, C_i \rangle$ request, the matchmaker is guaranteed to never process a MatchA$\langle j, C_j \rangle$ request in the future. Thus, every matchmaker in M_i has not processed a MatchA request in round j and never will. For a value to be chosen in round j, the proposer executing round j must first receive replies from $f + 1$ matchmakers, say M_j, in round j. But, M_i and M_j necessarily intersect, so this is impossible. Thus, no value has been or will be chosen in round j.

Case 2: $j = k$. In a given round, at most one value is proposed, let alone chosen. x is *the* value proposed in round k, so no other value could be chosen in round k.

Case 3: $j < k$. By induction, $P(k)$ states that no value other than x has been or will be chosen in any round less than k. This includes round j.

Finally, if k is -1, then we are in the same situation as in Case 1. No value has or will be chosen in a round $j < i$. □

Garbage Collection (How)

We've discussed how a proposer can change its round and introduce a new configuration. Now, we explain how to shut down old configurations. At the beginning of round i, a proposer p executes the Matchmaking phase and computes a set H_i of configurations in rounds less than i. The proposer then executes Phase 1 with the acceptors in these configurations. Assume H_i contains a configuration C_j for a round $j < i$. If we prematurely shut down the acceptors in C_j, then proposer p will get stuck in Phase 1, waiting for Phase1B messages from a quorum of nodes that have been shut down. Therefore, we cannot shut down the

acceptors in a configuration C_j until we are sure that the matchmakers will never again return C_j during the Matchmaking phase.

Thus, we extend Matchmaker Paxos to allow matchmakers to garbage collect configurations from their logs, ensuring that the garbage collected configurations will not be returned during any future Matchmaking phase. More concretely, a proposer p can now send a GarbageA$\langle i \rangle$ command to the matchmakers informing them to garbage collect all configurations in rounds less than i. When a matchmaker receives a GarbageA$\langle i \rangle$ message, it deletes log entry $L[j]$ for every round $j < i$. It then updates a garbage collection watermark w to the maximum of w and i and sends back a GarbageB$\langle i \rangle$ message to the proposer. See Algorithm 11.

Algorithm 11 Matchmaker Pseudocode (with GC). Changes to Algorithm 8 are underlined and shown in blue.

State: a log L indexed by round, initially empty
State: a garbage collection watermark w, initially 0

1: **upon** receiving GarbageA$\langle i \rangle$ from proposer p **do**
2: delete $L[j]$ for all $j < i$.
3: $w \leftarrow \max(w, i)$
4: send GarbageB$\langle i \rangle$ to p

5: **upon** receiving MatchA$\langle i, C_i \rangle$ from proposer p **do**
6: **if** $i < w$ or $\exists\ C_j$ in round $j \geq i$ in L **then**
7: ignore the MatchA$\langle i, C_i \rangle$ message
8: **else**
9: $H_i \leftarrow \{(j, C_j) \mid C_j \in L\}$
10: $L[i] \leftarrow C_i$
11: send MatchB$\langle i, w, H_i \rangle$ to p

We also update the Matchmaking phase in three ways. First, a matchmaker ignores a MatchA$\langle i, C_i \rangle$ message if i has been garbage collected (i.e. if $i < w$). Second, a matchmaker returns its garbage collection watermark w in every MatchB that it sends. Third, when a proposer receives MatchB$\langle i, w_1, H_i^1 \rangle$, ..., MatchB$\langle i, w_{f+1}, H_i^{f+1} \rangle$ from $f+1$ matchmakers, it again computes $H_i = \cup_{j=1}^{f+1} H_i^j$. It then computes $w = \max_{j=1}^{f+1} w_j$ and prunes every configuration in H_i in a round less than w. In other words, if any of the $f+1$ matchmakers have garbage collected round j, then the proposer also garbage collects round j.

Once a proposer receives GarbageB$\langle i \rangle$ messages from at least $f+1$ matchmakers M, it is guaranteed that all future Matchmaking phases will not include any configuration in any round less than i. Why? Consider a future Matchmaking phase run with $f+1$ matchmakers M'. M and M' intersect, so some matchmaker in the intersection has a garbage collection watermark at least as large as i. Thus, once a configuration has been garbage collected by $f+1$ matchmakers, we can shut down the acceptors in the configuration.

Garbage Collection (When)

Once a configuration has been garbage collected, it is safe to shut it down, but when is it safe to garbage collect a configuration? It is not always safe. For example, if we prematurely garbage collect configuration C_j in round j, a future proposer in round $i > j$ may not learn about a value v chosen in round j and then erroneously get a value other than v chosen in round i. There are three situations in which it is safe for a proposer p_i in round i to issue a GarbageA$\langle i \rangle$ command. We first explain the three situations and provide intuition on why they are safe before proving they are safe. Later, we'll see that all three scenarios are important for Matchmaker MultiPaxos.

Scenario 1. If the proposer p_i gets a value x chosen in round i, then it can safely issue a GarbageA$\langle i \rangle$ command. Why? When a proposer p_j in round $j > i$ executes Phase 1, it will learn about the value x and propose x in Phase 2. But first, it must establish that no value other than x has been or will be chosen in any round less than j. This is $P(j)$ from the safety proof in Section 6.1. The proposer p_i already established this fact for all rounds less than i (this is $P(i)$), so any communication with the configurations in these rounds is redundant. Thus, we can garbage collect them.

Scenario 2. If the proposer p_i executes Phase 1 in round i and finds $k = -1$ (see Algorithm 10), then it can safely issue a GarbageA$\langle i \rangle$ command. Recall that if $k = -1$, then no value has been or will be chosen in any round less than i. This situation is similar to Scenario 1. Any future proposer p_j in round $j > i$ does not have to redundantly communicate with the configurations in rounds less than i since p_i already established that no value has been chosen in these rounds.

Scenario 3. If the proposer p_i learns that a value x has already been chosen and has been stored on $f + 1$ non-acceptor machines (e.g., $f + 1$ proposers), then the proposer can safely issue a GarbageA$\langle i \rangle$ command after it informs a Phase 2 quorum of acceptors in C_i of this fact. Any future proposer p_j in round $j > i$ will contact a Phase 1 quorum of C_i and encounter at least one acceptor that knows the value x has already been chosen. When this acceptor informs p_j that a value x has already been chosen, p_j stops executing the protocol entirely and simply fetches the value x from one of the $f + 1$ machines that store the value. Note that storing the value on $f+1$ machines ensures that some machine will store the value despite f failures. The decision of exactly which $f + 1$ machines is not important.

To prove that these three scenarios are safe, we repeat the safety proof above. The new bits are shown in blue.

Proof. We prove, for every round i, the statement $P(i)$: "if a proposer proposes a value v in round i (i.e. sends a Phase2A message for value v in round i), then no value other than v has been or will be chosen in any round less than i." At most one value is ever proposed in a given round, so at most one value is ever chosen in a given round. Thus, $P(i)$ suffices to prove that Matchmaker Paxos is safe for the following reason. Assume for contradiction that Matchmaker Paxos chooses distinct values x and y in rounds j and i with $j < i$. Some

proposer must have proposed y in round i, so $P(i)$ ensures us that no value other than y could have been chosen in round j. But, x was chosen, a contradiction.

We prove $P(i)$ by strong induction on i. $P(0)$ is vacuous because there are no rounds less than 0. For the general case $P(i)$, we assume $P(0), \ldots, P(i-1)$. We perform a case analysis on the proposer's pseudocode (Algorithm 10). Either k is -1 or it is not (line 11). First, assume it is not. In this case, the proposer proposes x, the value proposed in round k (line 12). We perform a case analysis on round j to show that no value other than x has been or will be chosen in any round $j < i$.

Case 1: $j > k$. We show that no value has been or will be chosen in round j. Recall that at the end of the Matchmaking phase, the proposer computed the set H_i of prior configurations using responses from a set M_i of $f+1$ matchmakers. Either H_i contains a configuration C_j in round j or it doesn't.

First, suppose it does. Then, the proposer sent Phase1A$\langle i \rangle$ messages to all of the acceptors in C_j. A Phase 1 quorum of these acceptors, say Q, all received Phase1A$\langle i \rangle$ messages and replied with Phase1B messages. Thus, every acceptor in Q set its round r to i, and in doing so, promised to never vote in any round less than i. Moreover, none of the acceptors in Q had voted in any round greater than k. So, every acceptor in Q has not voted and never will vote in round j. For a value v' to be chosen in round j, it must receive votes from some Phase 2 quorum Q' of round j acceptors. But, Q and Q' necessarily intersect, so this is impossible. Thus, no value has been or will be chosen in round j.

Now suppose that H_i does *not* contain a configuration for round j. Either a configuration C_j was garbage collected from H_i or it wasn't. First, assume it wasn't. Then, H_i is the union of $f+1$ MatchB messages from the $f+1$ matchmakers in M_i. Thus, if H_i does not contain a configuration for round j, then none of the MatchB messages did either. This means that for every matchmaker $m \in M_i$, when m received MatchA$\langle i, C_i \rangle$, it did not contain a configuration for round j in its log and never did. Moreover, by processing the MatchA$\langle i, C_i \rangle$ request and inserting C_i in log entry i, the matchmaker is guaranteed to never process a MatchA$\langle j, C_j \rangle$ request in the future. Thus, every matchmaker in M_i has not processed a MatchA request in round j and never will. For a value to be chosen in round j, the proposer executing round j must first receive replies from $f+1$ matchmakers, say M_j, in round j. But, M_i and M_j necessarily intersect, so this is impossible. Thus, no value has been or will be chosen in round j.

Otherwise, a configuration C_j was garbage collected from H_i. Note that none of the matchmakers in M_i had received a GarbageA$\langle i' \rangle$ command for a round $i' > i$ when they responded with their MatchB messages. If they had, they would have ignored our MatchA$\langle i, C_i \rangle$ message. Let i' be the largest round $j < i' < i$ such that a matchmaker in M_i had received a GarbageA$\langle i' \rangle$ message before responding to our MatchA$\langle i, C_i \rangle$ message.

If i' was garbage collected because of Scenario 1, then k would be at least as large as i' since we would have intersected the Phase 2 quorum of $C_{i'}$ used in round i' to get a value chosen. But $k < j < i'$, a contradiction. If i' was garbage collected because of Scenario 2, then we know no value has been or will be chosen in round j. If i' was garbage collected because of Scenario 3, then we would have intersected the Phase 2 quorum of $C_{i'}$ that knows

a value was already chosen, and we would have not proposed a value in the first place. But, we proposed x, a contradiction.

Case 2: $j = k$**.** In a given round, at most one value is proposed, let alone chosen. x is *the* value proposed in round k, so no other value could be chosen in round k.

Case 3: $j < k$**.** By induction, $P(k)$ states that no value other than x has been or will be chosen in any round less than k. This includes round j.

Finally, if k is -1, then we are in the same situation as in Case 1. No value has or will be chosen in a round $j < i$. □

Later, we'll extend this garbage collection protocol to Matchmaker MultiPaxos (Section 6.2) and see empirically that matchmakers usually return just a single configuration (Section 6.5).

Optimizations

We now present a couple of protocol optimizations. First, note that a proposer can proactively run the Matchmaking phase in round i *before* it hears from a client. This is similar to proactively executing Phase 1, a standard optimization [29]. We call this optimization **proactive matchmaking**.

Second, assume that the proposer in round i has executed the Matchmaking phase and Phase 1. Through Phase 1, it finds that $k = -1$ and thus learns that no value has been chosen in any round less than i (see the safety proof above). Assume that before executing Phase 2 in round i, the proposer decides to perform a reconfiguration. To perform the reconfiguration, the proposer stops executing round i and begins executing the next round $i + 1$[1]. Typically to perform the reconfiguration, the proposer would have to execute the Matchmaking phase, Phase 1, and Phase 2 in round $i + 1$. However, in this case, after executing the Matchmaking phase in round $i+1$, the proposer can skip Phase 1 and proceed directly to Phase 2. Why? The proposer established in round i that no value has been or will be chosen in any round less than i. Moreover, because it did not run Phase 2 in round i, it also knows that no value has been or will be chosen in round i. Together, these imply that no value has been or will be chosen in any round less than $i+1$. Normally, the proposer would run Phase 1 in round $i+1$ to establish this fact, but since it has already established it, it can instead proceed directly to Phase 2. We call this optimization **Phase 1 bypassing**.

Phase 1 Bypassing depends on a proposer being the leader of round i *and* the leader of the next round $i + 1$. We can construct a set of rounds such that this is always the case. Let the set of rounds be the set of lexicographically ordered tuples (r, id, s) where r and s are both integers and id is a proposer id. A proposer is responsible for all the rounds that contain its id. With this set of rounds, the proposer p in round (r, p, s) always owns the next round $(r, p, s + 1)$. For example given two proposers a and b, we have the following ordering

[1]Note that given a round i, we denote the next largest round in the total ordered set of rounds $i + 1$. We call this the "next round".

on rounds:

$$
\begin{aligned}
(0, a, 0) < (0, a, 1) < (0, a, 2) < (0, a, 3) < \cdots \\
(0, b, 0) < (0, b, 1) < (0, b, 2) < (0, b, 3) < \cdots \\
(1, a, 0) < (1, a, 1) < (1, a, 2) < (1, a, 3) < \cdots
\end{aligned}
$$

We assume this round scheme throughout the rest of the chapter. In the next section, we'll see that this optimization is essential for implementing Matchmaker MultiPaxos with good performance. Also note that this optimization is not particular to Matchmaker Paxos. Paxos and MultiPaxos can both take advantage of this optimization.

6.2 Matchmaker MultiPaxos

MultiPaxos

In Section 2.3, we discussed the normal case execution of Paxos. Now, we elaborate on MultiPaxos leader changes and the execution of Phase 1. Assume a proposer is elected leader in some round, say round i. We assume the leader knows that log entries up to and including log entry k_c have already been chosen (e.g., by communicating with the replicas). We call this log entry the **commit index**. The leader then runs Phase 1 of Paxos in round i for *every* log entry. Note that even though there are an infinite number of log entries larger than k_c, the leader can execute Phase 1 using a finite amount of information. In particular, the leader sends a single Phase1A$\langle i \rangle$ message that acts as the Phase1A message for every log entry larger than k_c. Also, an acceptor replies with a Phase1B$\langle i, vr, vv \rangle$ message only for log entries in which the acceptor has voted. The infinitely many log entries in which the acceptor has not yet voted do not yield an explicit Phase1B message.

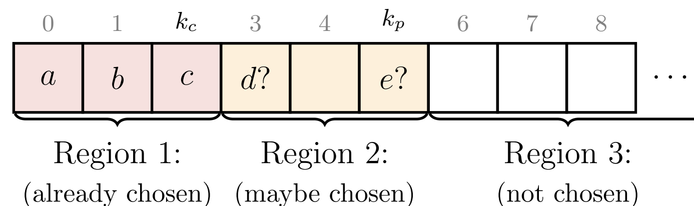


Figure 6.3: A leader's knowledge of the log after Phase 1.

The leader's knowledge about the log after Phase 1 can be characterized by the commit index k_c and a **pending index** k_p with $k_c \leq k_p$, as shown in Figure 6.3. The commit index and pending index divide the log into three regions: a prefix of chosen log entries (Region 1), a suffix of unchosen log entries (Region 3), and a middle region of pending log entries (Region 2). More specifically:

- **Region 1** $[0, k_c]$**:** The leader knows that a command has been chosen in every log entry less than or equal to k_c.
- **Region 3** $[k_p+1, \infty)$**:** The leader knows that no command has been chosen (in any round less than i) in any log entry larger than k_p.
- **Region 2** $[k_c+1, k_p]$**:** If there is a command that *may* have already been chosen, then it appears between k_c and k_p. Region 2 may also contain some log entries in which the leader knows (from executing a previous round) that a value has already been chosen, and it may contain some log entries in which the leader knows (from counting votes in Phase 1) that no value has been chosen (we call these "holes").

After Phase 1, the leader sends a PHASE2A message for every unchosen log entry in Region 2, proposing a "no-op" command for the holes. Simultaneously, the leader begins accepting client requests. When a client wants to propose a state machine command, it sends the command to the leader. The leader assigns log entries to commands in increasing order, beginning at k_p+1. It then runs Phase 2 of Paxos to get the command chosen in that entry in round i. Once the leader learns that a command has been chosen in a given log entry, it informs the replicas. Replicas insert chosen commands into their logs and execute the logs in prefix order, sending the results of execution back to the clients.

It is critical to note that a leader performs Phase 1 of Paxos only once *per round*, not once *per command*. In other words, Phase 1 is not performed during normal operation. It is performed only when the leader fails and a new leader is elected in a larger round, an uncommon occurrence.

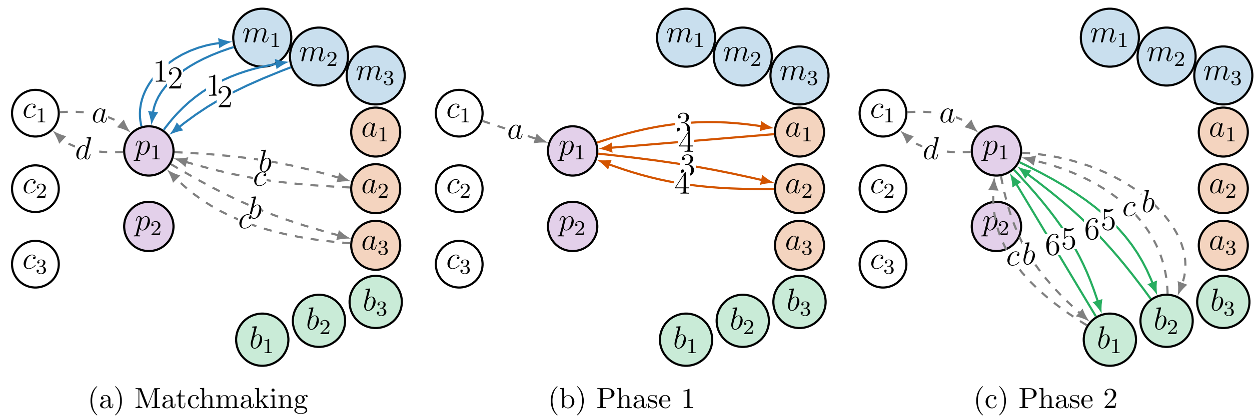
Matchmaker MultiPaxos

We first extend Matchmaker Paxos to Matchmaker MultiPaxos with proactive matchmaking but without Phase 1 bypassing or garbage collection. We'll see how to incorporate these two momentarily. The extension from Matchmaker Paxos to Matchmaker MultiPaxos is analogous to the extension of Paxos to MultiPaxos. Matchmaker MultiPaxos reaches consensus on a totally ordered log of state machine commands, one log entry at a time, using one instance of Matchmaker Paxos for every log entry.

More concretely, a Matchmaker MultiPaxos deployment consists of an arbitrary number of clients, at least $f+1$ proposers, a set of $2f+1$ matchmakers, a dynamic set of acceptors (one configuration per round which can tolerate f failures), and a set of at least $f+1$ state machine replicas. We assume, as is standard, that a leader election algorithm is used to select one of the proposers as a stable leader in some round, say round i. The leader selects a configuration C_i of acceptors that it will use for *every* log entry. The mechanism by which the configuration is chosen is an orthogonal concern. A system administrator, for example, could send the configuration to the leader, or the configuration could be read from an external service.

The leader then executes the Matchmaking phase in the same way as in Matchmaker Paxos (i.e. it sends $\textsc{MatchA}\langle i, C_i \rangle$ messages to the matchmakers and awaits $\textsc{MatchB}\langle i, H_i \rangle$ responses). After the Matchmaking phase completes, the leader executes Phase 1 for *every* log entry. This is identical to MultiPaxos, except that the leader uses the configurations returned by the matchmakers rather than assuming a fixed configuration. Note that proactive matchmaking allows the leader to execute the Matchmaking phase and Phase 1 before receiving any client requests.

The leader then enters Phase 2 and operates exactly as it would in MultiPaxos. It executes Phase 2 with C_i for the log entries in Region 2. Moreover, when it receives a state machine command from a client, it assigns the command a log entry in Region 3, runs Phase 2 with the acceptors in C_i, and informs the replicas when the command is chosen. Replicas execute commands in log order and send the results of executing commands back to the clients.



(a) Matchmaking (b) Phase 1 (c) Phase 2

Figure 6.4: An example Matchmaker MultiPaxos reconfiguration without Phase 1 bypassing. The leader p_1 reconfigures from the acceptors a_1, a_2, a_3 to the acceptors b_1, b_2, b_3. Client commands are drawn as gray dashed lines. Note that every subfigure shows one phase of a reconfiguration using solid colored lines, but the dashed lines show the complete execution of a client request that runs concurrently with the reconfiguration. For simplicity, we assume that every proposer also serves as a replica.

Discussion

To reconfigure from some old configuration C_{old} in round i to some new configuration C_{new}, the Matchmaker MultiPaxos leader of round i simply advances to round $i+1$ and selects the new configuration C_{new}. The new configuration is active immediately after the Matchmaking phase, a one round trip delay. Note that the acceptors in the new configuration C_{new} do not have to undergo any sort of warm up or bootstrapping and do not have to contact any other acceptors in any other configuration.

The new configuration is active immediately, but it is not safe to deactivate the acceptors in the old configuration immediately, as we saw in Section 6.1. We extend Matchmaker Paxos's garbage collection to Matchmaker MultiPaxos momentarily.

Also note that Matchmaker MultiPaxos does *not* perform the Matchmaking phase or Phase 1 on the critical path of normal execution. Similar to how MultiPaxos executes Phase 1 only once per leader change (and not once per command), Matchmaker MultiPaxos runs the Matchmaking phase and Phase 1 only when a new leader is elected or when a leader changes its round (e.g., when a leader transitions from round i to round $i + 1$ as part of a reconfiguration). In the normal case (i.e. during Phase 2), Matchmaker MultiPaxos and MultiPaxos are identical, and Matchmaker MultiPaxos does not introduce *any* overheads. In the normal case, Matchmaker MultiPaxos deploys a single stable leader that changes rounds only to perform a reconfiguration. Changing from one leader to another only happens after a leader has failed.

Furthermore, configurations do not have to be unique across rounds. The leader in round i is free to re-use a configuration C_j that was used in some round $j < i$.

Optimization

Ideally, Matchmaker MultiPaxos' performance would be unaffected by a reconfiguration. The latency of every client request and the protocol's overall throughput would remain constant throughout a reconfiguration. Matchmaker MultiPaxos as we've described it so far, however, does not meet this ideal. During a reconfiguration, a leader must temporarily stop processing client commands and wait for the reconfiguration to finish before resuming normal operation.

This is illustrated in Figure 6.4. Figure 6.4 shows a leader p_1 reconfiguring from a configuration of acceptors C_{old} consisting of acceptors a_1, a_2, and a_3 in round i to a new configuration of acceptors C_{new} consisting of acceptors b_1, b_2, and b_3 in round $i + 1$. While the leader performs the reconfiguration, clients continue to send state machine commands to the leader. We consider such a command and perform a case analysis on when the command arrives at the leader to see whether or not the command has to be stalled.

Case 1: Matchmaking (Figure 6.4a). If the leader receives a command during the Matchmaking phase, then the leader can process the command as normal in round i using the acceptors in C_{old}. Even though the leader is executing the Matchmaking phase in round $i+1$ and is communicating with the matchmakers, the acceptors in C_{old} are oblivious to this and can process commands in Phase 2 in round i.

Case 2: Phase 1 (Figure 6.4b). If the leader receives a command during Phase 1, then the leader cannot process the command. It must delay the processing of the command until Phase 1 finishes. Here's why. Once an acceptor in C_{old} receives a $\textsc{Phase1A}\langle i + 1\rangle$ message, it will reject any future commands in rounds less than $i+1$, so the leader is unable to send the command to C_{old}. The leader also cannot send the command to C_{new} in round $i + 1$ because it has not yet finished executing Phase 1.

Case 3: Phase 2 (Figure 6.4c). If the leader receives a command during Phase 2, then the leader can send the command to the new acceptors in C_{new} in round $i + 1$. This is

the normal case of execution.

In summary, any commands received during Phase 1 of a reconfiguration are delayed. Fortunately, we can eliminate this problem by using Phase 1 bypassing. Consider a leader performing a reconfiguration from C_i in round i to C_{i+1} in round $i+1$. At the end of the Matchmaking phase and at the beginning of Phase 1 (in round $i+1$), let k be the largest log entry that the leader has assigned to a command. That is, all log entries after entry k are empty. These log entries satisfy the preconditions of Phase 1 bypassing, so it is safe for the leader to bypass Phase 1 in round $i+1$ for these log entries in the following way. When a leader receives a command after the Matchmaking phase, it assigns the command a log entry larger than k, skips Phase 1, and executes Phase 2 in round $i+1$ with C_{new} immediately.

With this optimization and the round scheme described in Section 6.1, no state machine commands are delayed. Commands received during the Matchmaking phase or earlier are chosen in round i by C_{old} in log entries up to and including k. Commands received during Phase 1, Phase 2, or later are chosen in round $i+1$ by C_{new} in log entries $k+1$, $k+2$, $k+3$, and so on. With this optimization Matchmaker MultiPaxos can be reconfigured with minimal performance degradation.

Garbage Collection

Recall that the Matchmaker MultiPaxos leader p_i in round i uses a single configuration C_i for *every* log entry. The leader p_i can safely issue a GarbageA$\langle i \rangle$ command to the matchmakers once it ensures that *every* log entry satisfies one of the three scenarios described in Section 6.1. Recall from Figure 6.3 that at the end of Phase 1 and at the beginning of Phase 2, the log can be divided into three regions. Each of the three garbage collection scenarios applies to one of the regions.

Scenario 2 applies to Region 3. These are the log entries for which $k = -1$. Scenario 1 applies to Region 2, once the leader has successfully chosen commands in all of the log entries in Region 2. Scenario 3 applies to Region 1 if we make the following adjustments. First, we deploy $2f+1$ replicas instead of $f+1$. Second, the leader ensures that the prefix of previously chosen log entries is stored on at least $f+1$ of the $2f+1$ replicas. Third, the leader informs a Phase 2 quorum of C_i acceptors that these commands have been stored on the replicas. Every replica maintains a copy of the log of state machine commands and cannot discard a command after execution. The log must also be garbage collected over time, for example, by using snapshots [67]. Note that garbage collecting the log is an orthogonal (but also complicated) issue from garbage collecting old configurations. It must be done regardless of reconfigurations and is outside of the scope of this thesis.

In summary, the leader p_i of round i executes as follows. It executes the Matchmaking phase to get the prior configurations H_i. It executes Phase 1 with the configurations in H_i. It enters Phase 2 and chooses commands in Region 2. It informs a Phase 2 quorum of C_i acceptors once the commands in Region 1 have been stored on $f+1$ replicas. It issues a GarbageA$\langle i \rangle$ command to the matchmakers and awaits $f+1$ GarbageB$\langle i \rangle$ responses. At this point, all previous configurations can be shut down.

Note that the leader can begin processing state machine commands from clients as soon as it enters Phase 2. It does not have to stall commands during garbage collection. Note also that during normal operation, old configurations are garbage collected very quickly. In Section 6.5, we show that H_i almost always contains a single configuration (i.e. C_{i-1}).

6.3 Reconfiguring Matchmakers

We've discussed how Matchmaker MultiPaxos allows us to reconfigure the set of acceptors. In this section, we discuss how to reconfigure proposers, replicas, and matchmakers (themselves).

Reconfiguring proposers and replicas is straightforward. In fact, Matchmaker MultiPaxos reconfigures proposers and replicas in exactly the same way as MultiPaxos [87], so we do not discuss it at length. In short, a proposer can be safely added or removed at any time. Replicas can also be safely added or removed at any time so long as we ensure that commands replicated on $f + 1$ replicas remain replicated on $f + 1$ replicas. This is a difficult, yet orthogonal challenge. Existing approaches can be adopted by Matchmaker MultiPaxos [68]. For performance, a newly introduced proposer should contact an existing proposer or replica to learn about the prefix of already chosen commands, and a newly introduced replica should copy the log from an existing replica.

Reconfiguring matchmakers is a bit more involved, but still relatively straightforward. Recall that proposers perform the Matchmaking phase only during a change in round. Thus, for the vast majority of the time—specifically, when there is a single, stable leader—the matchmakers are completely idle. This means that the way we reconfigure the matchmakers has to be safe, but it doesn't have to be efficient. The matchmakers can be reconfigured at any time between round changes without any impact on the performance.

Thus, we use the simplest approach to reconfiguration: we shut down the old matchmakers and replace them with new ones, making sure that the new matchmakers' initial state is the same as the old matchmakers' final state. More concretely, we reconfigure from a set M_{old} of matchmakers to a new set M_{new} as follows. First, a proposer (or any other node) sends a StopA$\langle\rangle$ message to the matchmakers in M_{old}. When a matchmaker m_i receives a StopA$\langle\rangle$ message, it stops processing messages (except for other StopA$\langle\rangle$ messages) and replies with StopB$\langle L_i, w_i\rangle$ where L_i is m_i's log and w_i is its garbage collection watermark. When the proposer receives StopB messages from $f+1$ matchmakers, it knows that the matchmakers have effectively been shut down. It computes w as the maximum of every returned w_i. It computes L as the union of the returned logs, and removes all entries of L that appear in a round less than w. An example of this log merging is illustrated in Figure 6.5.

The proposer then sends L and w to all of the matchmakers in M_{new}. Each matchmaker adopts these values as its initial state. At this point, the matchmakers in M_{new} *cannot* begin processing commands yet. Naively, it is possible that two different nodes could simultaneously attempt to reconfigure to two disjoint sets of matchmakers, say M_{new} and M'_{new}.

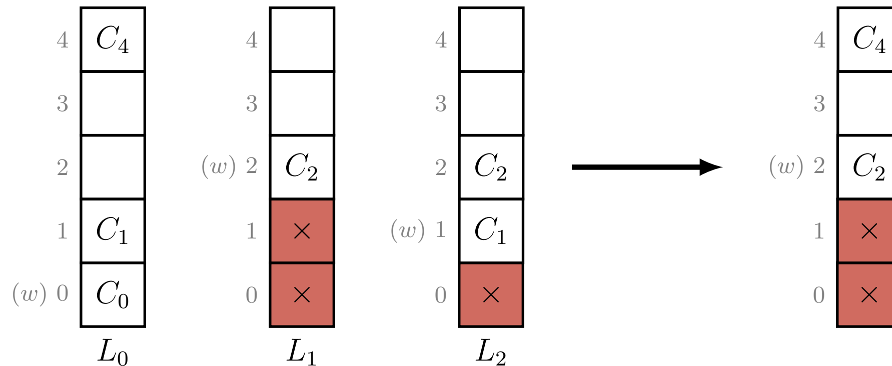


Figure 6.5: An example of merging three matchmaker logs (L_0, L_1, and L_2) during a matchmaker reconfiguration. Garbage collected log entries are shown in red.

To avoid this, every matchmaker in M_{old} doubles as a Paxos acceptor. A proposer attempting to reconfigure to M_{new} acts as a Paxos proposer and gets the value M_{new} chosen by the matchmakers (which are acting as Paxos acceptors). Once M_{new} is chosen, the proposer notifies the matchmakers in M_{new} that the reconfiguration is complete and that they are free to start processing commands.

If a proposer contacts a stale set of matchmakers (e.g., M_{old}), the matchmakers inform the proposer of their successors (e.g., M_{new}). This newer set of matchmakers may also be stale, so the proposer repeatedly polls stale matchmakers until it finds the active set of matchmakers. In this way, the matchmakers form a chain, with each set of matchmakers pointing to its successor.

Before a set of matchmakers can be shut down, the identity of its successors must be persisted in some name service (e.g., DNS). Ideally for performance, the name service would always point to the active set of matchmakers, but this is not required for safety.

We now prove that Matchmaker MultiPaxos is still safe, even with matchmaker reconfigurations. The new bits are shown in blue.

Proof. We prove, for every round i, the statement $P(i)$: "if a proposer proposes a value v in round i (i.e. sends a Phase2A message for value v in round i), then no value other than v has been or will be chosen in any round less than i." At most one value is ever proposed in a given round, so at most one value is ever chosen in a given round. Thus, $P(i)$ suffices to prove that Matchmaker Paxos is safe for the following reason. Assume for contradiction that Matchmaker Paxos chooses distinct values x and y in rounds j and i with $j < i$. Some proposer must have proposed y in round i, so $P(i)$ ensures us that no value other than y could have been chosen in round j. But, x was chosen, a contradiction.

We prove $P(i)$ by strong induction on i. $P(0)$ is vacuous because there are no rounds less than 0. For the general case $P(i)$, we assume $P(0), \ldots, P(i-1)$. We perform a case analysis on the proposer's pseudocode (Algorithm 10). Either k is -1 or it is not (line 11).

First, assume it is not. In this case, the proposer proposes x, the value proposed in round k (line 12). We perform a case analysis on round j to show that no value other than x has been or will be chosen in any round $j < i$.

Case 1: $j > k$. We show that no value has been or will be chosen in round j. Recall that at the end of the Matchmaking phase, the proposer computed the set H_i of prior configurations using responses from a set M_i of $f + 1$ matchmakers. Either H_i contains a configuration C_j in round j or it doesn't.

First, suppose it does. Then, the proposer sent Phase1A$\langle i \rangle$ messages to all of the acceptors in C_j. A Phase 1 quorum of these acceptors, say Q, all received Phase1A$\langle i \rangle$ messages and replied with Phase1B messages. Thus, every acceptor in Q set its round r to i, and in doing so, promised to never vote in any round less than i. Moreover, none of the acceptors in Q had voted in any round greater than k. So, every acceptor in Q has not voted and never will vote in round j. For a value v' to be chosen in round j, it must receive votes from some Phase 2 quorum Q' of round j acceptors. But, Q and Q' necessarily intersect, so this is impossible. Thus, no value has been or will be chosen in round j.

Now suppose that H_i does *not* contain a configuration for round j. Either a configuration C_j was garbage collected from H_i or it wasn't. First, assume it wasn't. Then, H_i is the union of $f + 1$ MatchB messages from the $f + 1$ matchmakers in M_i. Thus, if H_i does not contain a configuration for round j, then none of the MatchB messages did either. This means that for every matchmaker $m \in M_i$, when m received MatchA$\langle i, C_i \rangle$, it did not contain a configuration for round j in its log and never did. Moreover, no majority in any previous set of matchmakers contained a configuration in round j. If any majority did have a configuration in round j, then all subsequent matchmakers would as well since a set of matchmakers is initialized from a majority of the previous matchmakers. Moreover, by processing the MatchA$\langle i, C_i \rangle$ request and inserting C_i in log entry i, the matchmaker is guaranteed to never process a MatchA$\langle j, C_j \rangle$ request in the future. Moreover, no future set of matchmakers will either. A majority of matchmakers have a configuration in entry i, so all subsequent configurations will as well. Therefore, they will all reject a configuration in round j. Thus, every matchmaker in M_i has not processed a MatchA request in round j and never will. For a value to be chosen in round j, the proposer executing round j must first receive replies from $f+1$ matchmakers, say M_j, in round j. But, M_i and M_j necessarily intersect, so this is impossible. This argument holds for every set of matchmakers. Thus, no value has been or will be chosen in round j.

Otherwise, a configuration C_j was garbage collected from H_i. Note that none of the matchmakers in M_i had received a GarbageA$\langle i' \rangle$ command for a round $i' > i$ when they responded with their MatchB messages. If they had, they would have ignored our MatchA$\langle i, C_i \rangle$ message. Similarly, none of the matchmakers in M_i were initialized with a garbage collection watermark $w > i$. Let i' be the largest round $j < i' < i$ that a matchmaker in M_i garbage collected before responding to our MatchA$\langle i, C_i \rangle$ message.

If i' was garbage collected because of Scenario 1, then k would be at least as large as i' since we would have intersected the Phase 2 quorum of $C_{i'}$ used in round i' to get a value chosen. But $k < j < i'$, a contradiction. If i' was garbage collected because of Scenario 2,

then we know no value has been or will be chosen in round j. If i' was garbage collected because of Scenario 3, then we would have intersected the Phase 2 quorum of $C_{i'}$ that knows a value was already chosen, and we would have not proposed a value in the first place. But, we proposed x, a contradiction.

Case 2: $j = k$. In a given round, at most one value is proposed, let alone chosen. x is *the* value proposed in round k, so no other value could be chosen in round k.

Case 3: $j < k$. We can apply the inductive hypothesis to get $P(k)$ which states that no value other than x has been or will be chosen in any round less than k. This includes round j, which is exactly what we're trying to prove.

Finally, if k is -1, then we are in the same situation as in Case 1. □

6.4 Theoretical Insights

MultiPaxos To reconfigure from a set of nodes N to a new set of nodes N', a MultiPaxos leader gets the value N' chosen in the log at some index i. All commands in the log starting at position $i+\alpha$ are chosen using the nodes in N' instead of the nodes in N, where α is some configurable parameter. This protocol is called **Horizontal MultiPaxos**.

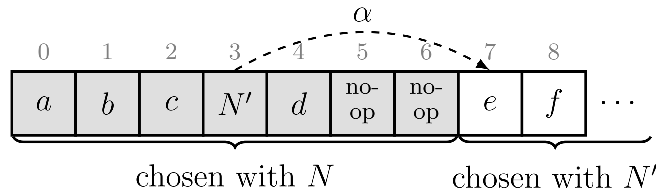


Figure 6.6: A MultiPaxos log during reconfiguration ($\alpha = 4$).

Matchmaker MultiPaxos has the following advantages over Horizontal MultiPaxos. First, the core idea behind Horizontal MultiPaxos seems simple, but the protocol has a number of hidden subtleties [54]. For example, a newly elected Horizontal MultiPaxos leader with a stale log may not know the latest configuration of nodes. It may not even know which configuration of nodes to contact to learn the latest configuration of nodes. This makes it unclear when it is safe to shut down old configurations because a newly elected Horizontal MultiPaxos leader can be arbitrarily out of date. These subtleties and the many others described in [54] makes Horizontal MultiPaxos significantly more complicated that it initially seems. Matchmaker Paxos addresses these subtleties directly. The matchmakers can always be used to learn the latest configuration, and our garbage collection protocol details exactly when and how to shut down old configurations safely.

Second, horizontal reconfiguration is not generally applicable. It is fundamentally incompatible with replication protocols that do not replicate a log. Moreover, researchers are finding that avoiding a log can often be advantageous [42, 62, 6, 74, 97, 82, 21]. For example,

protocols like Generalized Paxos [42], EPaxos [62], Atlas [21], and Caesar [6] arrange commands in a partially ordered graph instead of a totally ordered log to exploit commutativity between commands. CASPaxos [74] maintains a single value, instead of a log or graph, for simplicity. Databases like TAPIR [97] avoid ordering transactions in a log for improved performance, and databases like Meerkat [82] do the same to improve scalability. Even some protocols with logs cannot use the ideas behind Horizontal MultiPaxos. For example, Raft cannot safely perform Horizontal MultiPaxos' reconfiguration [67].

Because these protocols do not replicate logs, they cannot use MultiPaxos' horizontal reconfiguration protocol. However, while none of the protocols replicate logs, *all* of them have rounds. This means that the protocols can either use Matchmaker Paxos directly, or at least borrow ideas from Matchmaker Paxos for reconfiguration. For example, we are developing a protocol called BPaxos that is an EPaxos [62] variant which partially orders commands into a graph. BPaxos is a modular protocol that uses Paxos as a black box subroutine. Due to this modularity, we can directly replace Paxos with Matchmaker Paxos to support reconfiguration. The same idea can also be applied to EPaxos. CASPaxos [74] is similar to Paxos and can be extended to Matchmaker CASPaxos in the same way we extended Paxos to Matchmaker Paxos. These are two simple examples, and we don't claim that extending Matchmaker Paxos to some of the other more complicated protocols is always easy. But, the universality of rounds makes Matchmaker Paxos an attractive foundation on top of which other non-log based protocols can build their own reconfiguration protocols.

One could argue that these other protocols are not used as much in industry, so it's not that important for them to have reconfiguration protocols, but we think the causation is in the reverse direction! Without reconfiguration, these protocols cannot be used in industry.

Third, optimizing Horizontal MultiPaxos is not easy. A MultiPaxos leader can process at most α unchosen commands at a time. This makes α an important parameter to tune. If we set α too low, then we limit the protocol's pipeline parallelism and the throughput suffers. Note that a small α reduces the *normal case* throughput of Horizontal MultiPaxos, not just the throughput during reconfiguration. If we set α too high, then we have to wait a long time for a reconfiguration to complete. If we are reconfiguring because of a failed node, then we might have to endure a long reconfiguration with reduced throughput. Matchmaker MultiPaxos has no α parameter to tune. Note that Horizontal MultiPaxos can be implemented with an optimization in which we select a very large α and then get a sequence of α noops in the log to force a quick reconfiguration. This optimization helps avoid the difficulties of finding a good value of α, but the optimization introduces a new set of subtleties into the protocol. For example, the leader cannot process client requests while it is executing Phase 2 for the α noops. The protocol has to implement additional mechanisms to avoid this one round trip stall.

Foruth, Horizontal MultiPaxos requires a Phase 1 and Phase 2 quorum of acceptors from an old configuration in order to perform a reconfiguration after a leader failure, but Matchmaker MultiPaxos only requires a Phase 1 quorum. Some read optimized MultiPaxos variants perform reads against Phase 1 quorums [13]. These protocols benefit from having very small Phase 1 quorums and very large Phase 2 quorums, requiring Horizontal Multi-

Paxos to contact far more nodes that Matchmaker MultiPaxos during a reconfiguration.

Finally, we clarify that if Horizontal MultiPaxos is implemented with all of its subtleties ironed out, is deployed with a good choice of α, and is run with small Phase 2 quorums, then it can perform a reconfiguration without performance degradation. In this case, Horizontal MultiPaxos and Matchmaker MultiPaxos both reconfigure, in some sense, "optimally".

Vertical Paxos Matchmaker MultiPaxos significantly improves the practicality of Vertical Paxos [49] in a number of ways. First, Vertical Paxos is a consensus protocol, not a state machine replication protocol, and it's not easy to extend Vertical Paxos' garbage collection protocol to a state machine replication protocol. Vertical Paxos garbage collects old configurations in situations similar to Scenario 1 and Scenario 2 from Section 6.1. It does not include Scenario 3. Without this, old configurations cannot be garbage collected, which means that it is never safe to shut down old configurations.

Second, Vertical Paxos requires an external master but does not describe how to implement the master in an efficient way. We could implement the master using another state machine replication protocol like MultiPaxos, but this would be both slow and overly complex. Plus, we would have to implement a reconfiguration protocol for the master as well. Our matchmakers are analogous to the external master but show that such a master does not require a nested invocation of state machine replication.

Third, Vertical Paxos requires that a proposer execute Phase 1 in order to perform a reconfiguration. Thus, Vertical Paxos cannot be extended to MultiPaxos without causing performance degradation during reconfiguration. This is not the case for matchmakers thanks to Phase 1 bypassing.

Fourth, Vertical Paxos does not describe how proposers learn the configurations used in previous rounds and instead assumes that configurations are fixed in advance by an oracle. Matchmaker Paxos shows that this assumption is not necessary, as the matchmakers store every configuration.

Fast Paxos While Paxos quorums consist of $f + 1$ out of $2f + 1$ acceptors, Fast Paxos requires larger quorums. Many protocols have reduced Fast Paxos quorum sizes a bit, but to date, Fast Paxos quorum sizes have remained larger than classic Paxos quorum sizes [62, 21]. Using matchmakers, we can implement Fast Paxos with a fixed set of $f + 1$ acceptors (and hence with $f + 1$-sized quorums). Specifically, we deploy Fast Paxos with $f + 1$ acceptors, with a single unanimous Phase 2 quorum, and with singleton Phase 1 quorums.

Fast Paxos proposer pseudocode is given in Algorithm 12. We do not modify the Fast Paxos acceptor or the matchmakers. For simplicity, we assume that we deploy Fast Paxos with $f + 1$ acceptors, with a single unanimous Phase 2 quorum, and with singleton Phase 1 quorums. Generalizing to arbitrary configurations that satisfy Fast Paxos' quorum intersection requirements is straightforward. Note that Fast Paxos cannot leverage Phase 1 Bypassing. Also note while both MultiPaxos and our Fast Paxos variant both have quorums of size $f + 1$, our Fast Paxos variant has a *fixed* set of $f + 1$ acceptors, while MultiPaxos can

---

Algorithm 12 Fast Paxos Proposer Pseudocode

State: a round i, initially -1
State: the configuration C_i for round i, initially null
State: the prior configurations H_i for round i, initially null
1: $i \leftarrow$ next largest round owned by this proposer
2: $C_i \leftarrow$ an arbitrary configuration
3: send $\textsc{MatchA}\langle i, C_i \rangle$ to all of the matchmakers
4: **upon** receiving $\textsc{MatchB}\langle i, H_i^1 \rangle, \ldots, \textsc{MatchB}\langle i, H_i^{f+1} \rangle$ from $f+1$ matchmakers **do**
5: $\quad H_i \leftarrow \bigcup_{j=1}^{f+1} H_i^j$
6: $\quad$ send $\textsc{Phase1A}\langle i \rangle$ to every acceptor in H_i
7: **upon** receiving $\textsc{Phase1B}\langle i, -, - \rangle$ from a Phase 1 quorum from every configuration in H_i **do**
8: $\quad k \leftarrow$ the largest vr in any $\textsc{Phase1B}\langle i, vr, vv \rangle$
9: $\quad V \leftarrow$ the corresponding vv's in round k
10: $\quad$ **if** $k = -1$ **then**
11: $\quad\quad$ send $\textsc{Phase2A}\langle i, \mathsf{any} \rangle$ to every acceptor in C_i
12: $\quad$ **else if** $V = \{v\}$ **then**
13: $\quad\quad$ send $\textsc{Phase2A}\langle i, v \rangle$ to every acceptor in C_i
14: $\quad$ **else**
15: $\quad\quad$ send $\textsc{Phase2A}\langle i, \mathsf{any} \rangle$ to every acceptor in C_i

---

choose any set of $f+1$ acceptors from all $2f+1$ acceptors. This has some disadvantages in terms of tail latency and fault tolerance.

We now prove that our modifications to Fast Paxos are safe. For simplicity, we ignore garbage collection and matchmaker reconfiguration. Introducing those two features and proving them correct is pretty much identical to what we did with Matchmaker Paxos.

Proof. We prove, for every round i, the statement $P(i)$ which states that if a an acceptor votes for a value v in round i (i.e. sends a $\textsc{Phase2B}$ message for value v in round i), then no value other than v has been or will be chosen in any round less than i. $P(i)$ suffices to prove that Matchmaker Paxos is safe. Why? Well, assume for contradiction that Matchmaker Paxos chooses distinct values x and y in rounds i and j with $i < j$. Some acceptor must have voted for y in round j, so $P(j)$ ensures us that no value other than y could have been chosen in round i. But, x was chosen, a contradiction.

We prove $P(i)$ by strong induction on i. $P(0)$ is vacuous because there are no rounds less than 0. For the general case $P(i)$, we assume $P(0), \ldots, P(i-1)$. We perform a case analysis on the proposer's pseudocode. Either k is -1 or it is not (line 8). First, assume it is not. We perform a case analysis on rounds $j < i$.

Case 1: $j > k$**.** Recall that at the end of the Matchmaking phase, the proposer computed the set H_i of prior configurations using responses from a set M of $f+1$ matchmakers. Either H_i contains a configuration C_j in round j or it doesn't.

First, suppose it does. Then, the proposer sent PHASE1A$\langle i \rangle$ messages to all of the acceptors in C_j. A Phase 1 quorum of these acceptors, say Q, all received PHASE1A$\langle i \rangle$ messages and replied with PHASE1B messages. Thus, every acceptor in Q set its round r to i, and in doing so, promised to never vote in any round less than i. Moreover, none of the acceptors in Q had voted in any round greater than k. So, every acceptor in Q has not voted and never will vote in round j. For a value v' to be chosen in round j, it must receive votes from some Phase 2 quorum Q' of round j acceptors. But, Q and Q' necessarily intersect, so this is impossible. Thus, no value has been or will be chosen in round j.

Now suppose that H_i does *not* contain a configuration for round j. H_i is the union of $f+1$ MATCHB messages from the $f+1$ matchmakers in M. Thus, if H_i does not contain a configuration for round j, then none of the MATCHB messages did either. This means that for every matchmaker $m \in M$, when m received MATCHA$\langle i, C_i \rangle$, it did not contain a configuration for round j in its log. Moreover, by processing the MATCHA$\langle i, C_i \rangle$ request and inserting C_i in log entry i, the matchmaker is guaranteed to never process a MATCHA$\langle j, C_j \rangle$ request in the future. Thus, every matchmaker in M has not processed a MATCHA request in round j and never will. For a value to be chosen in round j, the proposer executing round j must first receive replies from $f+1$ matchmakers, say M', in round j. But, M and M' necessarily intersect, so this is impossible. Thus, no value has been or will be chosen in round j.

Case 2: $j = k$**.** If $V = \{v\}$, then the proposer proposes v. We must prove that no value other than v has been or will be chosen in round k. For a value to be chosen in round k, every acceptor must vote for it in round k. Some acceptor voted for v in round k, so it is the only value with the possibility of receiving a unanimous vote.

Otherwise V contains multiple distinct elements, and the proposer proposes `any`. We must prove that no value has been or will be chosen in round k. This is immediate since no value can receive a unanimous vote in round k, if two different values have received votes in round k.

Case 3: $j < k$**.** If $V = \{v\}$, then the proposer proposes v, and we must prove that no value other than v has been or will be chosen in any round less than k. This is immediate from $P(k)$. Otherwise, $V = \{v_1, v_2, \ldots\}$, and the proposer proposes `any`. We must prove that no value has been or will be chosen in any round less than k. $P(k)$ tells us that no value other than v_1 has been or will be chosen in any round less than k. $P(k)$ also tells us that no value other than v_2 has been or will be chosen in any round less than k. Thus, no value has been or will be chosen in any round less than k.

Finally, if k is -1, then we are in the same situation as in Case 1. No value has been or will be chosen in any round less than i. □

DPaxos DPaxos is a Paxos variant that allows every round to use a different subset of acceptors from some fixed set of acceptors. Matchmaker Paxos obviates the need for a fixed set of nodes. DPaxos' scope is limited to a single instance of consensus, whereas Matchmaker MultiPaxos shows how to efficiently reconfigure across multiple instances of

consensus simultaneously. We also discovered that DPaxos' garbage collection algorithm is unsafe. Matchmaker MultiPaxos fixes the bug.

Consider a DPaxos deployment with $f_d = 1$, $f_z = 0$, three zones, three nodes per zone, and delegate quorums. Thus, a replication quorum consists of two nodes in one zone, and a leader election quorum consists of two nodes in two zones. We name the nodes A through I. Beside each node, we display its ballot, vote ballot, vote value, and intents [44].

Zone 1	Zone 2	Zone 3
A $-1, -1, \bot, \emptyset$	D $-1, -1, \bot, \emptyset$	G $-1, -1, \bot, \emptyset$
B $-1, -1, \bot, \emptyset$	E $-1, -1, \bot, \emptyset$	H $-1, -1, \bot, \emptyset$
C $-1, -1, \bot, \emptyset$	F $-1, -1, \bot, \emptyset$	I $-1, -1, \bot, \emptyset$

Proposer 1 initiates the leader election phase in ballot 0 for value x. It selects $\{A, B, D, E\}$ as its leader election quorum and $\{B, C\}$ as its intent. It sends prepare messages to the leader election quorum, and the leader election quorum replies. Proposer 1 doesn't receive any intents, so it does not expand its leader election quorum. It also learns that no value has been chosen yet, so it proposes value x to B and C. Both accept the value.

Zone 1	Zone 2	Zone 3
A $0, -1, \bot, \{0 : \{B, C\}\}$	D $0, -1, \bot, \{0 : \{B, C\}\}$	G $-1, -1, \bot, \emptyset$
B $0, 0, x, \{0 : \{B, C\}\}$	E $0, -1, \bot, \{0 : \{B, C\}\}$	H $-1, -1, \bot, \emptyset$
C $0, 0, x, \emptyset$	F $-1, -1, \bot, \emptyset$	I $-1, -1, \bot, \emptyset$

Next, proposer 2 initiates the leader election phase in ballot 1 for value y. It selects $\{E, F, H, I\}$ as its leader election quorum and $\{G, H\}$ as its intent. It sends prepare messages to the leader election quorum, and the leader election quorum replies. Proposer 2 receives the intent $\{B, C\}$ in ballot 0 from E, so it expands its leader election quorum and sends a prepare message to C. Proposer 2 learns that value x was chosen in ballot 0, so it ditches y and proposes x to G and H. G accepts, but the propose message to H is dropped.

Zone 1	Zone 2	Zone 3
A $0, -1, \bot, \{0 : \{B, C\}\}$	D $0, -1, \bot, \{0 : \{B, C\}\}$	G $1, 1, x, \emptyset$
B $0, 0, x, \{0 : \{B, C\}\}$	E $1, -1, \bot, \{0 : \{B, C\}, 1 : \{G, H\}\}$	H $1, -1, \bot, \{1 : \{G, H\}\}$
C $0, 0, x, \{1 : \{G, H\}\}$	F $1, -1, \bot, \{1 : \{G, H\}\}$	I $1, -1, \bot, \{1 : \{G, H\}\}$

Next, garbage collection is run. The garbage collector contacts G and sees that it has accepted a value in ballot 1. It informs all the nodes to discard intents in ballots less than 1.

Zone 1	Zone 2	Zone 3
A $0, -1, \bot, \emptyset$	D $0, -1, \bot, \emptyset$	G $1, 1, x, \emptyset$
B $0, 0, x, \emptyset$	E $1, -1, \bot, \{1 : \{G, H\}\}$	H $1, -1, \bot, \{1 : \{G, H\}\}$
C $0, 0, x, \{1 : \{G, H\}\}$	F $1, -1, \bot, \{1 : \{G, H\}\}$	I $1, -1, \bot, \{1 : \{G, H\}\}$

Next, proposer 3 initiates the leader election phase in ballot 2 for value z It selects $\{D, E, H, I\}$ as its leader election quorum and $\{E, F\}$ as its intent. It sends prepare messages to the leader election quorum, and the leader election quorum replies. Proposer 3 receives intent $\{G, H\}$ in ballot 1, but has already included H in its leader election quorum, so it does not send any additional prepares. It learns that no value has been chosen (this is a bug, x was chosen), so it proposes value z to E and G. Both accept the value, and z is chosen. This is a bug since x was already chosen.

Zone 1	Zone 2	Zone 3
A $0, -1, \bot, \emptyset$	D $2, -1, \bot, \emptyset 2 : \{E, F\}$	G $1, 1, x, \emptyset$
B $0, 0, x, \emptyset$	E $2, 2, z, \{1 : \{G, H\}, 2 : \{E, F\}\}$	H $2, -1, \bot, \{1 : \{G, H\}, 2 : \{E, F\}\}$
C $0, 0, x, \{1 : \{G, H\}\}$	F $2, 2, z, \{1 : \{G, H\}\}$	I $2, -1, \bot, \{1 : \{G, H\}, 2 : \{E, F\}\}$

6.5 Evaluation

We now evaluate Matchmaker MultiPaxos. Matchmaker MultiPaxos is implemented in Scala using the Netty networking library. We deployed Matchmaker MultiPaxos on m5.xlarge AWS EC2 instances within a single availability zone. We deploy Matchmaker MultiPaxos with $f = 1$, $f + 1$ proposers, $2f + 1$ acceptors, $2f + 1$ matchmakers, and $2f + 1$ replicas. For simplicity, every node is deployed on its own machine, but in practice, nodes can be physically co-located. In particular, any two logical roles can be placed on the same machine, so long as the two roles are not the same. For example, we can co-locate a leader, an acceptor, a replica, and a matchmaker, but we can't co-locate two acceptors (without reducing the fault tolerance of the system). For simplicity, we deploy Matchmaker MultiPaxos with a trivial no-op state machine in which every state machine command is a one byte no-op. All of our results generalize to more complex state machines as well (the choice of state machine is orthogonal to reconfiguration).

Reconfiguration

Experiment Description. We run a benchmark with 1, 4, and 8 clients. Every client executes in a closed loop. It repeatedly proposes a state machine command, waits to receive a response, and then immediately proposes another command. Every benchmark runs for 35 seconds. During the first 10 seconds, we perform no reconfigurations. From 10 seconds to 20 seconds, the leader reconfigures the set of acceptors once every second. In practice, we would reconfigure much less often. This is a worst case stress test for Matchmaker MultiPaxos. For each of the ten reconfigurations, the leader selects a random set of $2f + 1$ acceptors from a pool of $2 \times (2f + 1)$ acceptors. At 25 seconds, we fail one of the acceptors. 5 seconds later, the leader performs a reconfiguration to replace the failed acceptor. The delay of 5 seconds is completely arbitrary. The leader can reconfigure sooner if desired.

We also perform this experiment with an implementation of MultiPaxos with horizontal reconfiguration. As with Matchmaker MultiPaxos, we deploy MultiPaxos with $f + 1$ proposers, $2f + 1$ acceptors, and $2f + 1$ replicas. We set α to 8. Because α is equal to the number of clients, MultiPaxos never stalls because of an insufficiently large α. We do not implement the noop optimization.

Results. The latency and throughput of Matchmaker MultiPaxos are shown in Figure 6.7. Throughput and latency are both computed using sliding one second windows. Median latency is shown using solid lines, while the 95% latency is shown as a shaded region above the median latency. The black vertical lines denote reconfigurations, and the red dashed vertical line denotes the acceptor failure.

The medians, interquartile ranges (IQR), and standard deviations of the latency and throughput (a) during the first 10 seconds and (b) between 10 and 20 seconds are shown in Table 6.1. Figure 6.10 includes violin plots of the same data. The white circles show the median values, while the thick black rectangles show the 25th and 75th percentiles. For latency, reconfiguration has little to no impact (roughly 2% changes) on the medians,

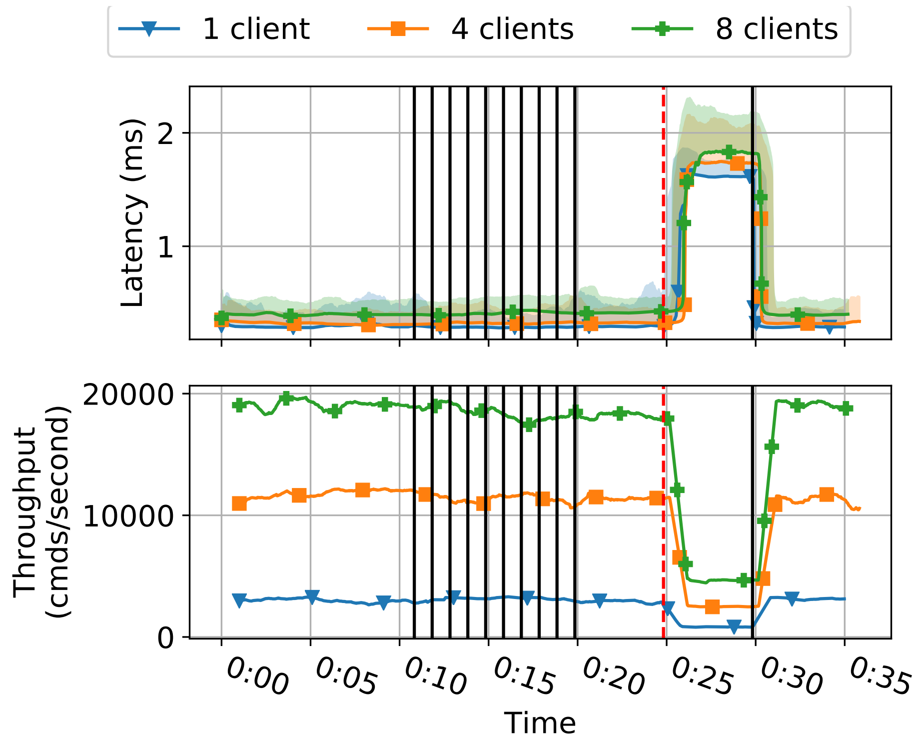


Figure 6.7: Matchmaker MultiPaxos' latency and throughput ($f = 1$). Median latency is shown using solid lines, while the 95% latency is shown as a shaded region above the median latency. The vertical black lines show reconfigurations. The vertical dashed red line shows an acceptor failure.

IQRs, or standard deviations. The one exception is that the 8 client standard deviation is significantly larger. This is due to a small number of outliers. Reconfiguration has little impact on median throughput, with all differences being statistically insignificant. The IQRs and standard deviations sometimes increase and sometimes decrease. The IQR is always less than 1% of the median throughput, and the standard deviation is always less than 4%.

For every reconfiguration, the new acceptors become active within a millisecond. The old acceptors are garbage collected within five milliseconds. This means that only one configuration is ever returned by the matchmakers. We implement Matchmaker MultiPaxos with an optimization called thriftiness [62]—where Phase2A messages are sent to a randomly selected Phase 2 quorum—so the throughput and latency expectedly degrade after we fail an acceptor. After we replace the failed acceptor, throughput and latency return to normal within two seconds.

The latency and throughput of MultiPaxos is shown in Figure 6.8. As with Matchmaker MultiPaxos, MultiPaxos can perform a horizontal reconfiguration without any performance degradation. The difference in absolute throughput between the two protocols is due to minor

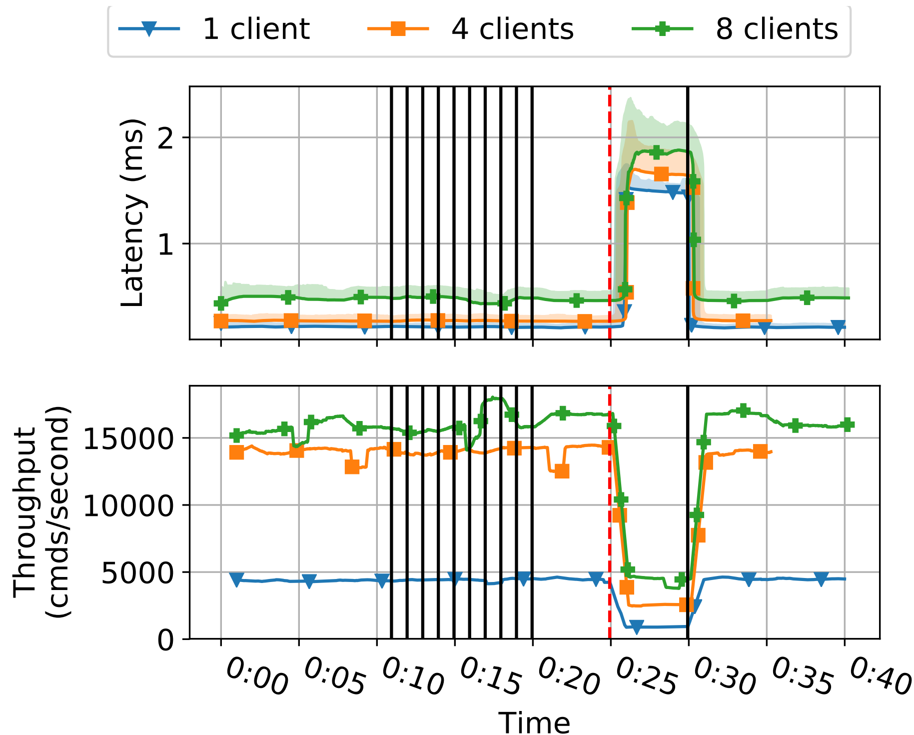


Figure 6.8: The latency and throughput of MultiPaxos with horizontal reconfiguration ($f = 1$).

implementation differences, but the variance in throughput (rather than the throughput itself) is what is important for this evaluation. We include the comparison to MultiPaxos for the sake of having some baseline against which we can compare Matchmaker MultiPaxos, but the comparison is shallow. For this reason, we do not elaborate on the results much.

While Matchmaker MultiPaxos does provide performance benefits over MultiPaxos' and Raft's reconfiguration protocols, our goal is not to replace these protocols. Rather, there are dozens of other state machine replication protocols (e.g., EPaxos [62], CASPaxos [74], Caesar [6], Atlas [21]) and distributed databases (e.g., TAPIR [97], Janus [63], Ocean Vista [22]) that do not have any reconfiguration protocol and cannot use the existing reconfiguration protocols from MultiPaxos or Raft. Our hope is that the ideas in Matchmaker MultiPaxos can be used to implement reconfiguration protocols for these other systems. For this reason, it is difficult to compare Matchmaker MultiPaxos against some existing baseline because they simply do not exist.

Summary. This experiment confirms that Matchmaker MultiPaxos's throughput and latency remain steady even during abnormally frequent reconfiguration. Moreover, it confirms that Matchmaker MultiPaxos can reconfigure to a new set of acceptors and retire the old set of acceptors on the order of milliseconds.

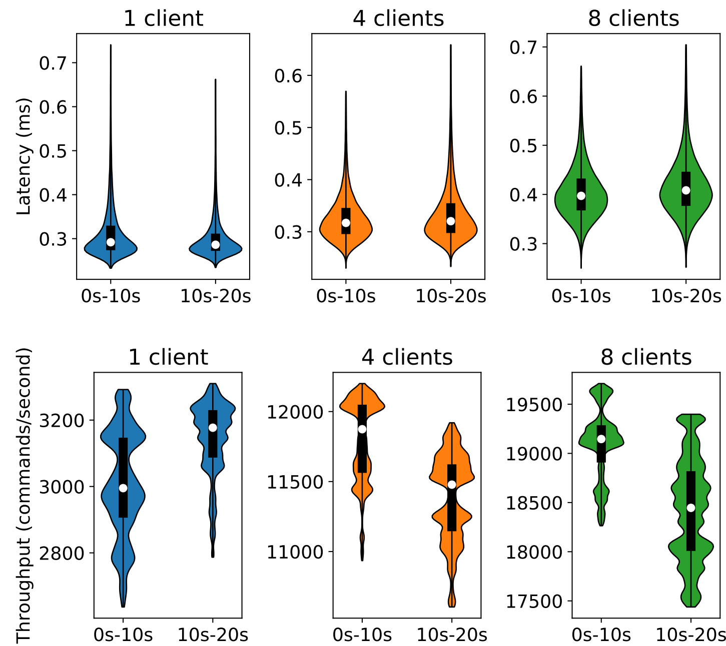


Figure 6.9: Violin plots of Figure 6.7 latency and throughput during the first 10 seconds and between 10 and 20 seconds.

Leader Failure

Experiment Description. We deploy Matchmaker MultiPaxos exactly as before. Now, each benchmark runs for 20 seconds. During the first 7 seconds, there are no reconfigurations and no failures. At 7 seconds, we fail the leader. 5 seconds later, a new leader is elected and resumes normal operation. The 5 second delay is arbitrary; a new leader could be elected quicker if desired.

Results. The latency and throughput of the benchmarks are shown in Figure 6.11. During the first 7 seconds, throughput and latency are both stable. When the leader fails, the throughput expectedly drops to zero. The throughput and latency return to normal within two seconds after a new leader is elected. The results for the same experiment, repeated with Horizontal MultiPaxos, are shown in Figure 6.12.

Summary. This experiment confirms that the extra latency of the Matchmaker phase during a leader change is negligible.

Matchmaker Reconfiguration

Experiment Description. We deploy Matchmaker MultiPaxos as above. We again run three benchmarks with 1, 4, and 8 clients. Each benchmark runs for 40 seconds. During the

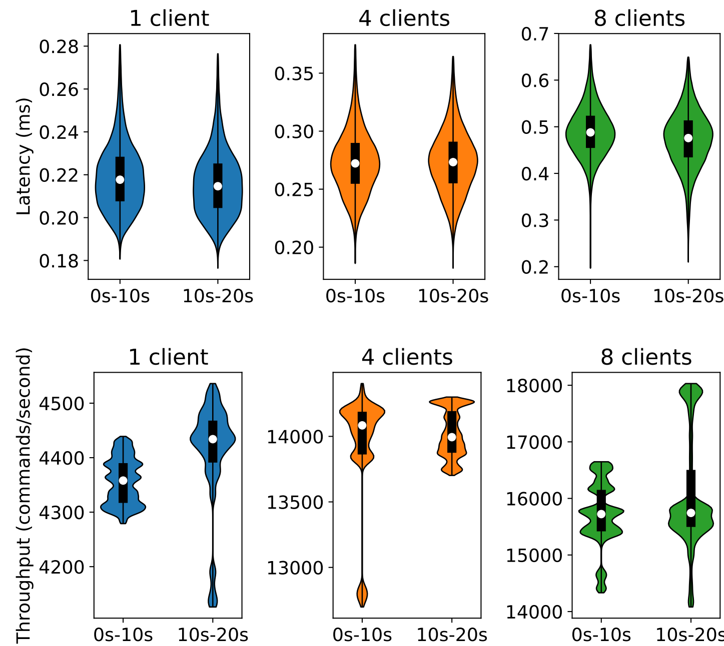


Figure 6.10: Violin plots of Figure 6.8 latency and throughput during the first 10 seconds and between 10 and 20 seconds.

first 10 seconds, there are no reconfigurations and no failures. Between 10 and 20 seconds, the leader reconfigures the set of matchmakers once every second. Every reconfiguration randomly selects $2f+1$ matchmakers from a set of $2 \times (2f+1)$ matchmakers. At 25 seconds, we fail a matchmaker. At 30 we perform a matchmaker reconfiguration to replace the failed matchmaker. At 35 seconds, we reconfigure the acceptors.

Results. The latency and throughput of Matchmaker MultiPaxos are shown in Figure 6.13. The latency and throughput of the protocol remain steady through the first ten matchmaker reconfigurations, through the matchmaker failure and recovery, and through the acceptor reconfiguration. This is confirmed by the medians, IQRs, and standard deviations of the latency and throughput during the first 10 seconds and between 10 and 20 seconds, which are shown in Table 6.2.

Summary. This benchmark confirms that matchmakers are off the critical path. The latency and throughput of Matchmaker MultiPaxos remains steady during a matchmaker reconfiguration and matchmaker failure. Moreover, a matchmaker reconfiguration does not affect the performance of subsequent acceptor reconfigurations.

Table 6.1: Figure 6.7 median, interquartile range, and standard deviation of latency and throughput.

Latency (ms)

	1 Client		4 Clients		8 Clients	
	0s-10s	10s-20s	0s-10s	10s-20s	0s-10s	10s-20s
median	0.292	0.287	0.317	0.321	0.398	0.410
IQR	0.040	0.026	0.029	0.036	0.036	0.039
stdev	0.114	0.085	0.076	0.081	0.089	0.305

Throughput (commands/second)

	1 Client		4 Clients		8 Clients	
	0s-10s	10s-20s	0s-10s	10s-20s	0s-10s	10s-20s
median	2,995	3,177	11,874	11,478	19,146	18,446
IQR	152	53	175	145	140	373
stdev	157	111	298	307	358	520

6.6 Related Work

SMART. SMART [54] is a reconfiguration protocol that resolves many ambiguities in MultiPaxos' horizontal approach (e.g., when it is safe to retire old configurations). Like MultiPaxos' horizontal reconfiguration protocol, SMART can reconfigure a protocol with minimal performance degradation. SMART differs from Matchmaker Paxos is a number of ways. First, like MultiPaxos' horizontal reconfiguration protocol, SMART is fundamentally log based and is therefore incompatible with many sophisticated state machine replication protocols. Second, SMART assumes that acceptors and replicas are always co-located. This prevents us from reconfiguring the acceptors without reconfiguring the replicas. This is not ideal since we can reconfigure an acceptor without copying any state, but must transfer logs from an old replica to a new replica. SMART's garbage collection also has higher latency that Matchmaker Paxos' garbage collection. For Scenario 3, Matchmaker Paxos proposers wait until a prefix of the log is stored on $f + 1$ replicas. SMART waits for the prefix of the log to be executed and snapshotted by $f + 1$ replicas.

Cheap Paxos. Cheap Paxos [50] is a MultiPaxos variant that consists of a fixed set of $f + 1$ main acceptors and f auxiliary acceptors. During failure-free execution (the normal case), only the main acceptors are contacted. The auxiliary acceptors perform MultiPaxos' horizontal reconfiguration protocol to replace failed main acceptors. As with Fast Paxos, we can deploy Matchmaker MultiPaxos with only $f + 1$ acceptors, f fewer than Cheap Paxos. Matchmaker Paxos does require $2f + 1$ matchmakers, but matchmakers do not act as acceptors and have to process only a single message (i.e. a MatchA message) to perform

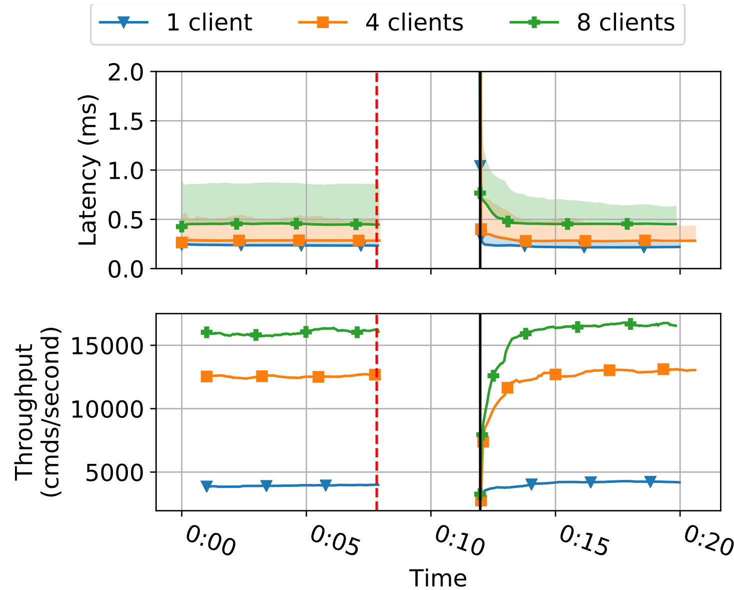


Figure 6.11: Matchmaker MultiPaxos' latency and throughput ($f = 1$). The dashed red line denotes a leader failure.

a reconfiguration.

Raft. Raft [68] uses a reconfiguration protocol called joint consensus. Like MultiPaxos' horizontal reconfiguration, joint consensus is log-based and therefore incompatible with many existing replication protocols. A simpler reconfiguration protocol for Raft was proposed in [67] but requires more rounds of communication.

Viewstamped Replication (VR). VR [53] uses a stop-the-world approach to reconfiguration. During a reconfiguration, the entire protocol stops processing commands. Thus, while the reconfiguration is quite simple, it is inefficient. Stoppable Paxos [48] is similar to MultiPaxos' horizontal reconfiguration, but also uses a stop-the-world approach. VR's stop-the-world approach to reconfiguration is also adopted by databases built on VR, including TAPIR [97] and Meerkat [82]. We use a similar approach to reconfigure matchmakers, but because matchmakers are off the critical path, the performance overheads are invisible.

Fast Paxos Coordinated Recovery. Fast Paxos has an optimization called coordinated recovery that is similar to Phase 1 Bypassing. The main difference is that in coordinated recovery, a leader uses Phase 2 information in round i to skip Phase 1 in round $i+1$, whereas with Phase 1 Bypassing, the leader instead uses Phase 1 information. Note that coordinated recovery is not useful for Matchmaker MultiPaxos. It is subsumed by Phase 1 Bypassing. Coordinated recovery is only needed for Fast Paxos because the leader may not know which values were proposed in a round it owns. Phase 1 Bypassing cannot be applied to Fast Paxos for pretty much the same reason.

DynaStore. Vertical Paxos assumes its external master is implemented using state ma-

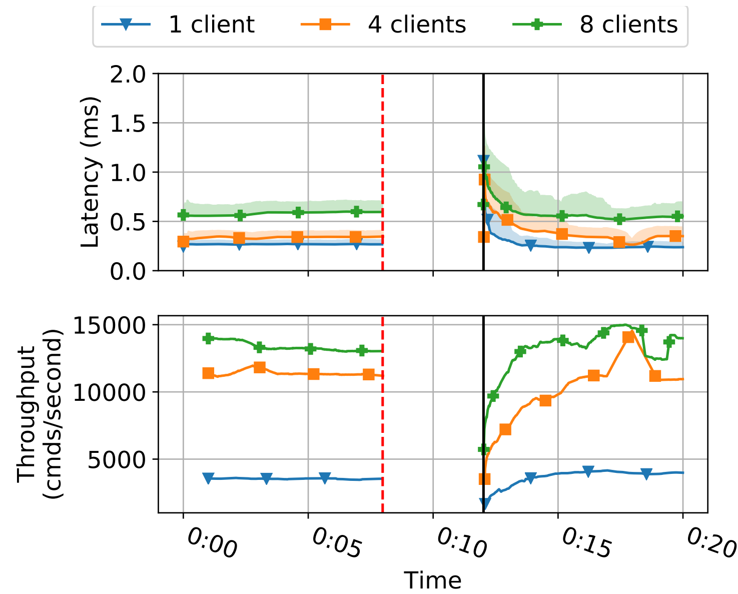


Figure 6.12: The latency and throughput of Horizontal MultiPaxos with $f = 1$.

chine replication. MultiPaxos' horizontal reconfiguration also depends on consensus. Matchmaker Paxos does not require consensus to implement matchmakers, but we are not the first to notice this. DynaStore [3] showed that reconfiguring atomic storage does not require consensus.

ZooKeeper. ZooKeeper, a distributed coordinated service, which uses ZooKeeper Atomic Broadcast [36] is a protocol similar to MultiPaxos that can also reconfigure quickly after leader failures.

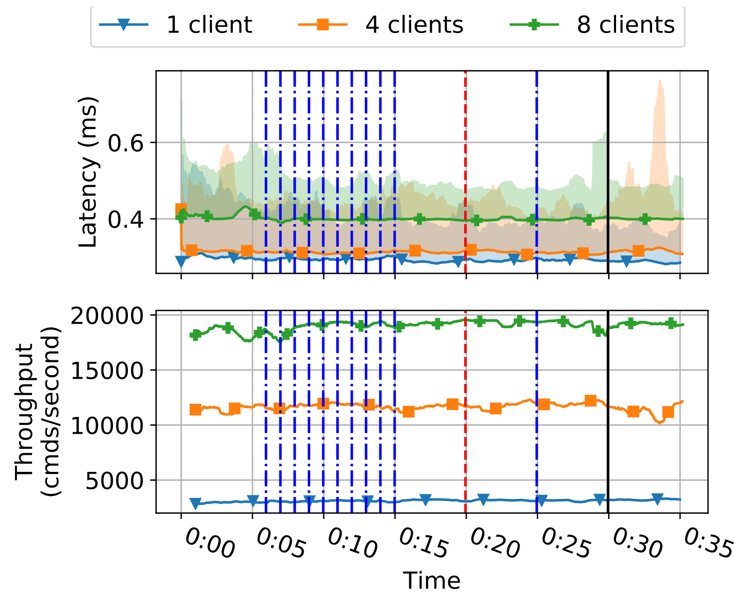


Figure 6.13: The latency and throughput of Matchmaker MultiPaxos ($f = 1$). The dotted blue, dashed red, and vertical black lines show matchmaker reconfigurations, a matchmaker failure, and an acceptor reconfiguration respectively.

Table 6.2: Figure 6.13 median, interquartile range, and standard deviation of latency and throughput.

Latency (ms)

	1 Client		4 Clients		8 Clients	
	0s-10s	10s-20s	0s-10s	10s-20s	0s-10s	10s-20s
median	0.297	0.292	0.314	0.313	0.404	0.398
IQR	0.032	0.024	0.031	0.030	0.035	0.028
stdev	0.077	0.061	0.093	0.098	0.383	0.067

Throughput (commands/second)

	1 Client		4 Clients		8 Clients	
	0s-10s	10s-20s	0s-10s	10s-20s	0s-10s	10s-20s
median	3019	3147	11631	11726	18569	19248
IQR	41	51	140	145	391	71
stdev	66	72	250	231	478	159

Chapter 7

Conclusion and Lessons Learned

In this thesis, we examined many facets of state machine replication. We compartmentalized MultiPaxos to increase its throughput by over 10×. We enhanced and automated the underlying theory of read-write quorum systems. We provided insight on a class of multi-leader generalized state machine replication protocols that were previously extremely challenging to understand. We designed a new reconfiguration protocol that provides theoretical insights and can be generalized to many protocols. Through this research, we have learned the following lessons.

State machine replication does not have to be slow The throughput of state machine replication protocols reported in the literature varies tremendously. You'll find papers with throughput in the hundreds [31], in the tens of thousands [61], in the tens of millions [98], and in the billions [35] of commands per second. So while state machine replication protocols like MultiPaxos *can* be slow if implemented in a straightforward way, our research and other existing research has shown that there are straightforward protocols that can be adopted and straightforward protocol optimizations that can be used to achieve high throughput state machine replication.

Practical aspects of consensus need more attention MultiPaxos is an old protocol and has received a lot of attention over many years, so naturally there are dozens of papers focusing on improving very small details in the protocol. How can we shrink quorums by one node? How can we increase the likelihood of taking the fast path? This research is both interesting and important, so its abundance is justified. Surprisingly though, there is very little research on many practical aspects of state machine replication. How can we reconfigure a protocol? How do we garbage collect a protocol? What do we have to store on disk and what can we store in memory? Answering these questions is vital for implementing state machine replication in practice and deserves more attention.

Throughput and latency are very different goals Many state machine replication protocols aim to achieve high throughput and low latency in both the local and wide area

network. However, we think that attempting to accomplish all these goals simultaneously is misguided. For example, we can use compartmentalization to achieve very high throughput within a single data center. In performing these optimizations, we increase commit time, but within a data center, this doesn't have a significant impact on latency. In the wide area, on the other hand, this increased commit time has a major impact on latency. In this setting, decreasing commit time is key. Unfortunately, decreasing commit time is a much much more challenging task that requires very clever protocols. Protocols like Fast Paxos [46], NOPaxos [51], SpecPaxos [71], and CURP [69] all reduce commit time and are all significantly more complicated than MultiPaxos. Ultimately, we believe that achieving high throughput within a data center is actually relatively straightforward, but achieving low latency in a wide area network is very complicated. Practitioners should pick the simplest protocol that meets their needs.

Complex protocols can be understood given the right format and framing Protocols like EPaxos [61], Caesar [6], and Atlas [21] are complex protocols. Explaining these protocols in a twelve page research paper is almost impossible. As a result, the protocols are presented tersely and many explanations and proofs are pushed to appendices and technical reports. The explanations are constrained by the publication format. Contrast this, for example, with Lamport's 36 page manuscript on Generalized Paxos [42]. Generalized Paxos is also a very complicated protocol, but Lamport is able to describe the protocol carefully. As another example, we believe our tutorial on generalized multi-leader protocols is the clearest and most comprehensive to date, but it's impossible to fit the tutorial in a standard twelve page paper. In summary, we believe that many protocols that are considered "too complicated" are more accurately "not able to be sufficiently explained given publishing constraints."

Automating compartmentalization The compartmentalizations performed in this dissertation were all done manually. Compartmentalization is intentionally straightforward to the point of being almost mechanical, which makes it an attractive target for automation. Distributed systems could be expressed in a high-level language like Bloom [5, 17] or TLA+ [96], and then a compiler could automatically identify opportunities for compartmentalization. The optimizer could also determine the appropriate amount of decoupling and scaling to meet a certain optimization objective.

Extending beyond state machine replication In this dissertation, we applied compartmentalization to state machine replication, but compartmentalization is a generally applicable technique and can be applied to many other types of protocols. For example, there is ongoing work to apply compartmentalization to Byzantine state machine replication protocols. Compartmentalization could also be applied to distributed model training, distributed data processing systems, distributed databases, and so on.

Bibliography

[1] *A Brief Introduction of TiDB*. https://pingcap.github.io/blog/2017-05-23-perconalive17/. Accessed: 2019-10-21.

[2] Divyakant Agrawal and Amr El Abbadi. "The Tree Quorum Protocol: An Efficient Approach for Managing Replicated Data". In: *Proceedings of the 16th International Conference on Very Large Data Bases*. VLDB '90. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990, pp. 243–254. ISBN: 155860149X.

[3] Marcos K Aguilera et al. "Dynamic atomic storage without consensus". In: *Journal of the ACM (JACM)* 58.2 (2011), pp. 1–32.

[4] Ailidani Ailijiang et al. "WPaxos: Wide Area Network Flexible Consensus". In: *IEEE Transactions on Parallel and Distributed Systems* (2019).

[5] Peter Alvaro et al. "Consistency Analysis in Bloom: a CALM and Collected Approach". In: *CIDR*. Citeseer. 2011, pp. 249–260.

[6] Balaji Arun et al. "Speeding up Consensus by Chasing Fast Decisions". In: *2017 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*. IEEE. 2017, pp. 49–60.

[7] Berk Atikoglu et al. "Workload analysis of a large-scale key-value store". In: *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*. 2012, pp. 53–64.

[8] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. "Consensus-oriented parallelization: How to earn your first million". In: *Proceedings of the 16th Annual Middleware Conference*. ACM. 2015, pp. 173–184.

[9] Carlos Eduardo Bezerra, Fernando Pedone, and Robbert Van Renesse. "Scalable state-machine replication". In: *2014 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*. IEEE. 2014, pp. 331–342.

[10] Martin Biely et al. "S-paxos: Offloading the leader for high throughput state machine replication". In: *2012 IEEE 31st Symposium on Reliable Distributed Systems (SRDS)*. IEEE. 2012, pp. 111–120.

[11] Mike Burrows. "The Chubby lock service for loosely-coupled distributed systems". In: *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*. 2006, pp. 335–350.

[12] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. "Paxos made live: an engineering perspective". In: *Proceedings of the 2007 ACM Symposium on Principles of Distributed Computing*. ACM. 2007, pp. 398–407.

[13] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. "Linearizable quorum reads in Paxos". In: *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*. 2019.

[14] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. "PigPaxos: Devouring the communication bottlenecks in distributed consensus". In: *Proceedings of the 2021 International Conference on Management of Data*. ACM. 2021, pp. 235–247.

[15] Shun Yan Cheung, Mustaque Ahamad, and Mostafa H. Ammar. "Multidimensional voting: a general method for implementing synchronization in distributed systems". In: *Proceedings of the 10th International Conference on Distributed Computing Systems*. May 1990, pp. 362–369. DOI: 10.1109/ICDCS.1990.89304.

[16] Shun Yan Cheung, Mostafa H. Ammar, and Mustaque Ahamad. "The Grid Protocol: A High Performance Scheme for Maintaining Replicated Data". In: *IEEE Trans. on Knowl. and Data Eng.* 4.6 (Dec. 1992), pp. 582–592. ISSN: 1041-4347. DOI: 10.1109/69.180609. URL: https://doi.org/10.1109/69.180609.

[17] Neil Conway et al. "Logic and lattices for distributed programming". In: *Proceedings of the Third ACM Symposium on Cloud Computing*. 2012, pp. 1–14.

[18] James C Corbett et al. "Spanner: Google's globally distributed database". In: *ACM Transactions on Computer Systems (TOCS)* 31.3 (2013), p. 8.

[19] Giuseppe DeCandia et al. "Dynamo: Amazon's highly available key-value store". In: *ACM SIGOPS operating systems review* 41.6 (2007), pp. 205–220.

[20] Cong Ding et al. "Scalog: Seamless Reconfiguration and Total Order in a Scalable Shared Log". In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 2020, pp. 325–338.

[21] Vitor Enes et al. "State-machine replication for planet-scale systems". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020, pp. 1–15.

[22] Hua Fan and Wojciech Golab. "Ocean vista: gossip-based visibility control for speedy geo-distributed transactions". In: *Proceedings of the VLDB Endowment* 12.11 (2019), pp. 1471–1484.

[23] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. "Impossibility of distributed consensus with one faulty process". In: *Journal of the ACM (JACM)* 32.2 (1985), pp. 374–382.

[24] Hector Garcia-Molina and Daniel Barbara. "How to Assign Votes in a Distributed System". In: *J. ACM* 32.4 (Oct. 1985), pp. 841–860. ISSN: 0004-5411. DOI: 10.1145/4221.4223. URL: https://doi.org/10.1145/4221.4223.

[25] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google file system". In: *Proceedings of the 19th Symposium on Operating Systems Principles*. ACM. 2003, pp. 29–43.

[26] David K. Gifford. "Weighted Voting for Replicated Data". In: *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*. SOSP '79. New York, NY, USA: Association for Computing Machinery, 1979, pp. 150–162. ISBN: 0897910095. DOI: `10.1145/800215.806583`. URL: `https://doi.org/10.1145/800215.806583`.

[27] *Global data distribution with Azure Cosmos DB - under the hood*. `https://docs.microsoft.com/en-us/azure/cosmos-db/global-dist-under-the-hood`. Accessed: 2019-10-21.

[28] Maurice P Herlihy and Jeannette M Wing. "Linearizability: A correctness condition for concurrent objects". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12.3 (1990), pp. 463–492.

[29] Heidi Howard. "Distributed consensus revised". PhD thesis. University of Cambridge, 2019.

[30] Heidi Howard, Aleksey Charapko, and Richard Mortier. "Fast Flexible Paxos: Relaxing Quorum Intersection for Fast Paxos". In: *International Conference on Distributed Computing and Networking 2021*. 2021, pp. 186–190.

[31] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. "Flexible Paxos: Quorum Intersection Revisited". In: *20th International Conference on Principles of Distributed Systems (OPODIS 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2017.

[32] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. "Flexible Paxos: Quorum Intersection Revisited". In: *20th International Conference on rinciples of Distributed Systems (OPODIS 2016)*. Ed. by Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone. Vol. 70. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 25:1–25:14. ISBN: 978-3-95977-031-6. DOI: `10.4230/LIPIcs.OPODIS.2016.25`. URL: `http://drops.dagstuhl.de/opus/volltexte/2017/7094`.

[33] Heidi Howard and Richard Mortier. "Paxos vs Raft: Have we reached consensus on distributed consensus?" In: *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*. 2020, pp. 1–9.

[34] Toshihide Ibaraki and Tiko Kameda. "A theory of coteries: Mutual exclusion in distributed systems". In: *IEEE Transactions on Parallel and Distributed Systems* 4.7 (1993), pp. 779–794.

[35] Xin Jin et al. "Netchain: Scale-free sub-rtt coordination". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 2018, pp. 35–49.

[36] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. "Zab: High-performance broadcast for primary-backup systems". In: *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*. IEEE. 2011, pp. 245–256.

[37] Manos Kapritsos and Flavio Paiva Junqueira. “Scalable Agreement: Toward Ordering as a Service”. In: *HotDep*. 2010.

[38] Kubernetes. *Kubernetes*. `https://kubernetes.io`. accessed: 2020-03-01.

[39] Akhil Kumar. “Hierarchical Quorum Consensus: A New Algorithm for Managing Replicated Data”. In: *IEEE Trans. Comput.* 40.9 (Sept. 1991), pp. 996–1004. ISSN: 0018-9340. DOI: `10.1109/12.83661`. URL: `https://doi.org/10.1109/12.83661`.

[40] Avinash Lakshman and Prashant Malik. “Cassandra: a decentralized structured storage system”. In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40.

[41] Leslie Lamport. “Fast paxos”. In: *Distributed Computing* 19.2 (2006), pp. 79–103.

[42] Leslie Lamport. “Generalized consensus and Paxos”. In: (2005).

[43] Leslie Lamport. “How to make a multiprocessor computer that correctly executes multiprocess programm”. In: *IEEE Transactions on Computers* 9 (1979), pp. 690–691.

[44] Leslie Lamport. “Paxos made simple”. In: *ACM Sigact News* 32.4 (2001), pp. 18–25.

[45] Leslie Lamport. “The part-time parliament”. In: *ACM Transactions on Computer Systems (TOCS)* 16.2 (1998), pp. 133–169.

[46] Leslie Lamport. “Time, clocks, and the ordering of events in a distributed system”. In: *Concurrency: the Works of Leslie Lamport*. 2019, pp. 179–196.

[47] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. “Reconfiguring a state machine.” In: *SIGACT News* 41.1 (2010), pp. 63–73.

[48] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. “Stoppable paxos”. In: *TechReport, Microsoft Research* (2008).

[49] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. “Vertical paxos and primary-backup replication”. In: *Proceedings of the 28th ACM symposium on Principles of distributed computing*. 2009, pp. 312–313.

[50] Leslie Lamport and Mike Massa. “Cheap paxos”. In: *International Conference on Dependable Systems and Networks, 2004*. IEEE. 2004, pp. 307–314.

[51] Jialin Li et al. “Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering”. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, pp. 467–483.

[52] *Lightweight transactions in Cassandra 2.0*. `https://www.datastax.com/blog/2013/07/lightweight-transactions-cassandra-20`. Accessed: 2019-10-21.

[53] Barbara Liskov and James Cowling. “Viewstamped replication revisited”. In: (2012).

[54] Jacob R Lorch et al. “The SMART way to migrate replicated stateful services”. In: *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*. 2006, pp. 103–115.

[55] Giuliano Losa, Sebastiano Peluso, and Binoy Ravindran. "Brief announcement: A family of leaderless generalized-consensus algorithms". In: *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*. ACM. 2016, pp. 345–347.

[56] Mamoru Maekawa. "A square root N Algorithm for Mutual Exclusion in Decentralized Systems". In: *ACM Trans. Comput. Syst.* 3.2 (May 1985), pp. 145–159. ISSN: 0734-2071. DOI: 10.1145/214438.214445. URL: https://doi.org/10.1145/214438.214445.

[57] Yanhua Mao, Flavio P Junqueira, and Keith Marzullo. "Mencius: building efficient replicated state machines for WANs". In: *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*. 2008, pp. 369–384.

[58] Parisa Jalili Marandi et al. "Ring Paxos: A high-throughput atomic broadcast protocol". In: *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*. IEEE. 2010, pp. 527–536.

[59] David Mazieres. "Paxos made practical". In: *Unpublished manuscript, Jan* (2007).

[60] Iulian Moraru, David G Andersen, and Michael Kaminsky. *A proof of correctness for Egalitarian Paxos*. Tech. rep. Technical report, Parallel Data Laboratory, Carnegie Mellon University, 2013.

[61] Iulian Moraru, David G Andersen, and Michael Kaminsky. "Paxos quorum leases: Fast reads without sacrificing writes". In: *Proceedings of the ACM Symposium on Cloud Computing*. 2014, pp. 1–13.

[62] Iulian Moraru, David G Andersen, and Michael Kaminsky. "There is more consensus in egalitarian parliaments". In: *Proceedings of the 24th Symposium on Operating Systems Principles*. ACM. 2013, pp. 358–372.

[63] Shuai Mu et al. "Consolidating concurrency control and consensus for commits under conflicts". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, pp. 517–532.

[64] Moni Naor and Avishai Wool. "The Load, Capacity, and Availability of Quorum Systems". In: *SIAM Journal on Computing* 27.2 (1998). ISSN: 0097-5397. DOI: 10.1137/S0097539795281232. URL: https://doi.org/10.1137/S0097539795281232.

[65] Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. "Dpaxos: Managing data closer to users for low-latency and mobile applications". In: *Proceedings of the 2018 International Conference on Management of Data*. ACM. 2018, pp. 1221–1236.

[66] Rajesh Nishtala et al. "Scaling memcache at facebook". In: *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 2013, pp. 385–398.

[67] Diego Ongaro. "Consensus: Bridging theory and practice". PhD thesis. Stanford University, 2014.

[68] Diego Ongaro and John K Ousterhout. "In search of an understandable consensus algorithm". In: *USENIX Annual Technical Conference*. 2014, pp. 305–319.

[69] Seo Jin Park and John Ousterhout. "Exploiting commutativity for practical fast replication". In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 2019, pp. 47–64.

[70] David Peleg and Avishai Wool. "Crumbling Walls: A Class of Practical and Efficient Quorum Systems". In: *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '95. Ottowa, Ontario, Canada: Association for Computing Machinery, 1995, pp. 120–129. ISBN: 0897917103. DOI: 10.1145/224964.224978. URL: https://doi.org/10.1145/224964.224978.

[71] Dan RK Ports et al. "Designing Distributed Systems Using Approximate Synchrony in Data Center Networks". In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 2015, pp. 43–57.

[72] *Raft Replication in YugaByte DB*. https://www.yugabyte.com/resources/raft-replication-in-yugabyte-db/. Accessed: 2019-10-21.

[73] Sajjad Rizvi, Bernard Wong, and Srinivasan Keshav. "Canopus: A scalable and massively parallel consensus protocol". In: *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*. 2017, pp. 426–438.

[74] Denis Rystsov. "CASPaxos: Replicated State Machines without logs". In: *arXiv preprint arXiv:1802.07000* (2018).

[75] Nuno Santos and André Schiper. "Achieving high-throughput state machine replication in multi-core systems". In: *2013 IEEE 33rd International Conference on Distributed Computing Systems*. Ieee. 2013, pp. 266–275.

[76] Nuno Santos and André Schiper. "Optimizing Paxos with batching and pipelining". In: *Theoretical Computer Science* 496 (2013), pp. 170–183.

[77] Nuno Santos and André Schiper. "Tuning paxos for high-throughput with batching and pipelining". In: *International Conference on Distributed Computing and Networking*. Springer. 2012, pp. 153–167.

[78] Fred B Schneider. "Implementing fault-tolerant services using the state machine approach: A tutorial". In: *ACM Computing Surveys (CSUR)* 22.4 (1990), pp. 299–319.

[79] William Schultz, Tess Avitabile, and Alyson Cabral. "Tunable Consistency in MongoDB". In: *Proceedings of the VLDB Endowment* 12.12 (2019), pp. 2071–2081.

[80] Rong Shi and Yang Wang. "Cheap and available state machine replication". In: *2016 USENIX Annual Technical Conference (USENIXATC 16)*. 2016, pp. 265–279.

[81] Pierre Sutra and Marc Shapiro. "Fast genuine generalized consensus". In: *2011 IEEE 30th Symposium on Reliable Distributed Systems (SRDS)*. IEEE. 2011, pp. 255–264.

[82] Adriana Szekeres et al. "Meerkat: Multicore-Scalable Replicated Transactions Following the Zero-Coordination Principle". In: *Proceedings of the Fourteenth EuroSys Conference 2020*. 2020.

[83] Rebecca Taft et al. "CockroachDB: The Resilient Geo-Distributed SQL Database". In: *Proceedings of the 2020 International Conference on Management of Data*. ACM. 2020, pp. 1493–1509.

[84] Hatem Takruri et al. "FLAIR: Accelerating Reads with Consistency-Aware Network Routing". In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 2020, pp. 723–737.

[85] Jeff Terrace and Michael J Freedman. "Object Storage on CRAQ: High-Throughput Chain Replication for Read-Mostly Workloads". In: *USENIX Annual Technical Conference*. June. San Diego, CA. 2009, pp. 1–16.

[86] Alexander Thomson et al. "Calvin: fast distributed transactions for partitioned database systems". In: *Proceedings of the 2012 International Conference on Management of Data*. ACM. 2012, pp. 1–12.

[87] Robbert Van Renesse and Deniz Altinbuken. "Paxos made moderately complex". In: *ACM Computing Surveys (CSUR)* 47.3 (2015), p. 42.

[88] Robbert Van Renesse and Fred B Schneider. "Chain Replication for Supporting High Throughput and Availability". In: *6th USENIX Symposium on Operating Systems Design and Implementation (OSDI 04)*. Vol. 4. 91–104. 2004.

[89] Matt Welsh, David Culler, and Eric Brewer. "SEDA: an architecture for well-conditioned, scalable internet services". In: *ACM SIGOPS Operating Systems Review*. Vol. 35. 5. ACM. 2001, pp. 230–243.

[90] Michael Whittaker and Joseph M Hellerstein. "Checking Invariant Confluence, In Whole or In Parts". In: *ACM SIGMOD Record* 49.1 (2020), pp. 7–14.

[91] Michael Whittaker and Joseph M Hellerstein. "Interactive checks for coordination avoidance". In: *Proceedings of the VLDB Endowment* 12.1 (2018), pp. 14–27.

[92] Michael Whittaker et al. "Debugging distributed systems with why-across-time provenance". In: *Proceedings of the ACM Symposium on Cloud Computing*. 2018, pp. 333–346.

[93] Michael Whittaker et al. "Online template induction for machine-generated emails". In: *Proceedings of the VLDB Endowment* 12.11 (2019), pp. 1235–1248.

[94] Michael Whittaker et al. "Read-Write Quorum Systems Made Practical". In: *Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data*. 2021, pp. 1–8.

[95] Michael Whittaker et al. *Scaling Replicated State Machines with Compartmentalization [Technical Report]*. 2020. arXiv: `2012.15762 [cs.DC]`.

[96] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. "Model checking TLA+ specifications". In: *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. Springer. 1999, pp. 54–66.

[97] Irene Zhang et al. “Building consistent transactions with inconsistent replication”. In: *ACM Transactions on Computer Systems (TOCS)* 35.4 (2018), p. 12.

[98] Hang Zhu et al. “Harmonia: Near-linear scalability for replicated storage with in-network conflict detection”. In: *Proceedings of the VLDB Endowment* 13.3 (2019), pp. 376–389.