CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Apollo

Distributed consensus revised

Heidi Howard

University of Cambridge
Computer Laboratory
Pembroke College

September 2018

This dissertation is submitted for
the degree of Doctor of Philosophy

Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the Preface and specified in the text.

It is not substantially the same as any that I have submitted, or, is being concurrently submitted for a degree or diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. I further state that no substantial part of my dissertation has already been submitted, or, is being concurrently submitted for any such degree, diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text

This dissertation does not exceed the regulation length of 60 000 words, including tables and footnotes.

Distributed consensus revised

Heidi Howard

Summary

We depend upon distributed systems in every aspect of life. Distributed consensus, the ability to reach agreement in the face of failures and asynchrony, is a fundamental and powerful primitive for constructing reliable distributed systems from unreliable components.

For over two decades, the Paxos algorithm has been synonymous with distributed consensus. Paxos is widely deployed in production systems, yet it is poorly understood and it proves to be heavyweight, unscalable and unreliable in practice. As such, Paxos has been the subject of extensive research to better understand the algorithm, to optimise its performance and to mitigate its limitations.

In this thesis, we re-examine the foundations of how Paxos solves distributed consensus. Our hypothesis is that these limitations are not inherent to the problem of consensus but instead specific to the approach of Paxos. The surprising result of our analysis is a substantial weakening to the requirements of this widely studied algorithm. Building on this insight, we are able to prove an extensive generalisation over the Paxos algorithm.

Our revised understanding of distributed consensus enables us to construct a diverse family of algorithms for solving consensus; covering classical as well as novel algorithms to reach consensus where it was previously thought impossible. We will explore the wide reaching implications of this new understanding, ranging from pragmatic optimisations to production systems to fundamentally novel approaches to consensus, which achieve new tradeoffs in performance, scalability and reliability.

Acknowledgments

First, and foremost, I would like to sincerely thank my advisor, Jon Crowcroft, for his tireless enthusiasm and support since our paths first crossed six years ago. Without his encouragement, I would not have believed it possible for to me to pursue a PhD.

It takes a village to raise a child and it takes a research group to raise a graduate student. Since my first steps into the world of research, the Systems Research Group (SRG) at the Computer Lab has been my home. I am indebted to my mentor, Richard Mortier, for his counsel and patience, particularly during the longer than expected final stretch of my PhD. I am also indebted to Tim Harris for insightful feedback on this thesis which substantially helped toward improving the clarity and readability. I am thankful for the friendship of my former fellow students, Natacha Crooks, Malte Schwarzkopf, Matthew Grosvenor, Shehar Bano and my current fellow students, Krittika D'Silva, Zahra Tarkhani, Mohibi Hussain and Marco Caballero. In addition to those already mentioned, I am sincerely grateful to my friends beyond the SRG, especially Laura Scriven, Shreedipta Mitra and Jeunese Payne, for keeping me sane over the years. I am grateful to my colleague Martin Kleppmann for our discussions on distributed systems during key points of this PhD.

I am deeply grateful to my 2nd supervisor, Anil Madhavapeddy, for adopting me into the OCaml labs community and regularly feeding me. Over the years, I have enjoyed many lively lunchtime discussions and trips to the food park with Gemma Gordon, David Allsopp and my office mates, KC Sivaramakrishnan and Stephen Dolan. I am also thankful to my former colleagues, Mindy Preston and Amir Chaudhry for their companionship during their time with me at the computer lab.

I have been extremely privileged to work with Dahlia Malkhi, my mentor in the field of distributed systems. Thanks are also due to the friends, new and old, who kept me company during my time in California; Nick Spooner, Diego Ongaro, Jenny Wolochow and Igor Zablotchi. Without you all my time away from Cambridge could have been very lonely.

Last but not least, I'd like to thank my husband, Olly Andrade, for the joy you have brought to my life over the last seven years as well as the countless hours spend proofreading this thesis. This thesis is dedicated to my father, Daniel Howard, who raised me and supported me throughout my life but sadly passed away after a short battle with cancer before the completion of this PhD.

Contents

1 Introduction . . . **13**
1.1 State of the art . . . 14
1.2 Historical background . . . 14
1.3 Motivation . . . 16
1.4 Approach . . . 17
1.5 Contributions . . . 18
1.5.1 Publications . . . 19
1.5.2 Follow up research . . . 19
1.6 Scope & limitations . . . 19

2 Consensus & Classic Paxos . . . **21**
2.1 Preliminaries . . . 21
2.1.1 Single acceptor algorithm . . . 23
2.2 Classic Paxos . . . 27
2.2.1 Proposer algorithm . . . 29
2.2.2 Acceptor algorithm . . . 31
2.3 Examples . . . 32
2.4 Properties . . . 38
2.5 Non-triviality . . . 39
2.6 Safety . . . 39
2.7 Progress . . . 45
2.8 Summary . . . 47

3 Known revisions . . . **49**
3.1 Negative responses (NACKs) . . . 49
3.2 Bypassing phase two . . . 52
3.3 Termination . . . 55
3.4 Distinguished proposer . . . 59
3.5 Phase ordering . . . 60
3.6 Multi-Paxos . . . 60
3.7 Roles . . . 61
3.8 Epochs . . . 62
3.9 Phase one voting for epochs . . . 63

3.10 Proposal copying . . . 65
3.11 Generalisation to quorums . . . 68
3.12 Miscellaneous . . . 69
3.13 Summary . . . 73

4 Quorum intersection revised . . . 75
4.1 Quorum intersection across phases . . . 75
4.1.1 Algorithm . . . 76
4.1.2 Safety . . . 76
4.1.3 Examples . . . 78
4.2 Quorum intersection across epochs . . . 78
4.2.1 Algorithm . . . 81
4.2.2 Safety . . . 81
4.2.3 Examples . . . 83
4.3 Implications . . . 85
4.3.1 Bypassing phase two . . . 86
4.3.2 Co-location of proposers and acceptors . . . 86
4.3.3 Multi-Paxos . . . 88
4.3.4 Voting for epochs . . . 90
4.4 Summary . . . 90

5 Promises revised . . . 91
5.1 Intuition . . . 91
5.2 Algorithm . . . 92
5.3 Safety . . . 92
5.4 Examples . . . 94
5.5 Summary . . . 96

6 Value selection revised . . . 97
6.1 Epoch agnostic algorithm . . . 97
6.1.1 Safety . . . 101
6.1.2 Progress . . . 104
6.1.3 Examples . . . 105
6.2 Epoch dependent algorithm . . . 106
6.2.1 Safety . . . 108
6.2.2 Progress . . . 109
6.3 Summary . . . 111

7 Epochs revised . . . 113
7.1 Epochs from an allocator . . . 114
7.2 Epochs by value mapping . . . 115

7.3 Epochs by recovery . . . 118
7.3.1 Intuition . . . 118
7.3.2 Algorithm . . . 120
7.3.3 Safety . . . 122
7.3.4 Progress . . . 125
7.3.5 Examples . . . 126
7.4 Hybrid epoch allocation . . . 134
7.4.1 Multi-path Paxos using allocator . . . 135
7.4.2 Multi-path Paxos using value-based allocation . . . 136
7.4.3 Multi-path Paxos using recovery . . . 136
7.5 Summary . . . 137

8 Conclusion . . . 141
8.1 Motivation . . . 141
8.2 Summary of contributions . . . 142
8.3 Implications of contributions . . . 143
8.3.1 Greater flexibility . . . 143
8.3.2 New progress guarantees . . . 144
8.3.3 Improved performance . . . 145
8.3.4 Better clarity . . . 146

Bibliography . . . 147

List of Figures

2.1 Single acceptor algorithm (Alg. 1,2) 25
2.2 Classic Paxos with serial proposers, p_1 then p_2 (Alg. 3,4) 33
2.3 Classic Paxos with serial proposers, p_2 then p_1 (Alg. 3,4) 34
2.4 Classic Paxos with a failed proposer, proposal is not recovered (Alg. 3,4) . 35
2.5 Classic Paxos with a failed proposer, proposal is recovered (Alg. 3,4) 36
2.6 Classic Paxos with duelling concurrent proposers (Alg. 3,4) 37

3.1 Classic Paxos with NACKs (Alg. 5,6) 52
3.2 Classic Paxos with bypass (Alg. 4,7) 53
3.3 Classic Paxos with termination (Alg. 8,9) 58
3.4 Classic Paxos with voting (Alg. 3,10) 64
3.5 Classic Paxos with proposal copying from NACKs (Alg. 4,11) 68
3.6 Classic Paxos with non-majority quorums (Alg. 4,12) 70

4.1 Paxos revision A with serial proposers (Alg. 4,13) 79
4.2 Paxos revision A with concurrent proposers (Alg. 4,13) 80
4.3 Paxos revision B with successfully phase one skipping (Alg. 4,14) 84
4.4 Paxos revision B with unsuccessfully phase one skipping (Alg. 4,14) 85

5.1 Paxos revision C with early phase one completion (Alg. 4,15) 95

7.1 Paxos for binary consensus (Alg. 23,22) 117
7.2 Epochs by recovery with simple serial proposers (1) (Alg. 24,29) 131
7.3 Epochs by recovery with simple serial proposers (2) (Alg. 24,29) 132
7.4 Epochs by recovery with simple serial proposers (3) (Alg. 24,29) 133
7.5 Epochs by recovery with concurrent identical proposals (Alg. 24,29) 134

List of Tables

2.1 Reference table of notation. 24
2.2 Messages exchanged in Classic Paxos 29
2.3 Use of algorithm properties to prove the safety of Classic Paxos 44
2.4 Comparison between SAA & Classic Paxos 47

4.1 Example quorums for All aboard Paxos with three acceptors $U = \{a_1, a_2, a_3\}$. 87
4.2 Alternative phase one *while* conditions 90

5.1 Simplified *while* conditions for line 6, Algorithm 15. 94

7.1 Examples of the counting quorums for epochs by recovery 130
7.2 Approaches to epoch allocation 134

List of Algorithms

1 Proposer algorithm for SAA 25
2 Acceptor algorithm for SAA 25
3 Proposer algorithm for Classic Paxos 30
4 Acceptor algorithm for Classic Paxos 31

5 Proposer algorithm for Classic Paxos with NACKs 50
6 Acceptor algorithm for Classic Paxos with NACKs 51
7 Proposer algorithm for Classic Paxos with phase two bypass 54
8 Proposer algorithm for Classic Paxos with termination 56
9 Acceptor algorithm for Classic Paxos with termination 57
10 Acceptor algorithm for Classic Paxos with voting 63
11 Proposer algorithm for Classic Paxos with proposal copying 66
12 Proposer algorithm for Classic Paxos with generalised quorums 71

13 Proposer algorithm for Paxos revision A 77
14 Proposer algorithm for Paxos revision B 82

15 Proposer algorithm for Paxos revision C. 93

16 Proposer algorithm for Revision A using *possibleValues*. 98
17 Classic algorithm for *possibleValues*. 99
18 Quorum-based algorithm for *possibleValues*. 100
19 Proposer algorithm for Revision B/C using *possibleValues*. 106
20 Quorum-based algorithm for *possibleValues* (Revision B/C). 107

21 Allocator algorithm 114
22 Proposer algorithm for binary decision 116

23 Acceptor algorithm for binary decision 118

24 Acceptor algorithm for epochs by recovery. 120

25 Proposer algorithm for Revision A with epochs by recovery. 121

26 Algorithm for *possibleValues* with epochs by recovery (Revision A). 122

27 Proposer algorithm with epochs by recovery and a fixed quorum. 126

28 Proposer algorithm with epochs by recovery and fixed quorums. 128

29 Proposer algorithm with epochs by recovery and counting quorums. 129

30 Phase zero of Multi-path Paxos with an allocator 135

31 Fast path - Proposer algorithm for Multi-path Paxos with recovery 137

32 Slow path - Proposer algorithm for Multi-path Paxos with recovery 138

Chapter 1

Introduction

We depend upon computer systems in every aspect of life. We expect systems to respond quickly, to behave as expected, and to be available when needed. However, the components which make up these systems, such as computers and the networks which connect them, are not reliable. Distributed consensus is the problem of how to reliably reach agreement in the face of failures and asynchrony. This longstanding challenge is fundamental to distributed systems and, when solved, gives us the power to construct reliable distributed systems from unreliable components.

Lamport's Paxos algorithm [Lam98] has been synonymous with distributed consensus for two decades [Mal]. It is widely deployed in production, and has been the subject of extensive research to optimise, extend and better understand the algorithm. Despite its popularity, Paxos performs poorly in practice, its inflexible approach is heavyweight, unscalable and can be unavailable in the face of asynchrony and failures.

This thesis re-examines the problem of distributed consensus and how we approach it. Firstly, we prove that Paxos is, in fact, one point on a broad spectrum of approaches to solving distributed consensus, opening the door to a new breed of performant, scalable and resilient consensus algorithms. Then, we explore some of the new algorithms made possible by this result; some of which are even able to achieve consensus where it was previously thought impossible.

In the next section, we describe the de facto approach to consensus in modern distributed systems (§1.1). For readers who are unfamiliar with the field, we then outline the historical context of this research, focusing on how early formulations of the problem of distributed consensus shaped (and arguably limited) how it is solved today (§1.2). This is followed by our critiques of this widely adopted approach and thus our motivation for re-examining how we can solve consensus (§1.3). Next, we describe the methodology we chose for the investigation into consensus (§1.4) and highlight the surprising contributions made to the field of consensus as a result (§1.5).

1.1 State of the art

The ability to reach an agreement between parties is a fundamental necessity of modern society, whether it is deciding a time for a meeting or whom will govern a country. The same is true for distributed computer systems where agreement is necessary for hosts to share consistent state for vital functions such as addressing, resource allocation, file systems, primary election, routing, locking, ordering and coordination.

Agreement covers a broad spectrum of decision problems in distributed systems. Distributed consensus is one such problem which is characterised by two guarantees: firstly that all decisions are final, without assuming reliability or synchrony (*safety guarantee*) and secondly that eventually a decision will be reached (*progress guarantee*). It is known to be impossible to guarantee progress without making assumptions regarding synchrony or reliability [FLP85]. Therefore, algorithms which solve consensus aim to guarantee progress under the weakest liveness assumptions possible.

The Paxos algorithm, originally proposed by Leslie Lamport in 1998 [Lam98] and later refined [Lam01a], is at the heart of how we achieve distributed consensus today[1]. Broadly speaking, its approach operates in two stages, each requiring agreement by the majority of participants. The first stage establishes one of the participants as the *leader*, preventing past leaders from making any further decisions. Once the majority of participants have agreed on who will lead, the leader proceeds to the second stage where decisions are made by getting the backing of the majority of participants. The leader is responsible for ensuring that all past decisions, learned during the first stage of the algorithm, are preserved and only proposes new values if it is safe to do so. This algorithm is guaranteed to reach a decision provided that at least a majority of participants are up and communicating synchronously. This approach is now widely adopted as the foundation of many production systems.

1.2 Historical background

The problem of distributed consensus emerged in the academic literature in the early 1980s. Originally, distributed consensus was a generalisation of a widely studied transaction commit problem from the field of distributed databases. Somewhat surprisingly, the problem of distributed consensus was popularised by a proof of its impossibility. Fischer, Lynch and Paterson [FLP85] demonstrated in 1985 that it is not possible for any distributed consensus algorithm to guarantee termination in an asynchronous system where participants may fail. The proof is notable for the surprisingly strong model under which

[1] For now, we use the term *Paxos* to refer to the algorithm as it is commonly used today, instead of as it was first described by Lamport. Often the term Multi-degree Paxos or just Multi-Paxos is used for this purpose.

it holds; namely reliable exactly-once out-of-order message delivery, a bound of at most one participant failing and agreement over a single binary value. This is known as the FLP result.

Now that it has been established that some assumptions regarding synchrony were necessary to guarantee termination of any distributed consensus algorithm, the question naturally arises of what these assumptions are and what are the weakest possible assumptions. These questions were considered by works such as Dolev, Dwork and Stockmeyer [DDS87] and Dwork, Lynch and Stockmeyer [DLS88]. The difficulty of reaching distributed consensus lies with the inability to reliably detect failures. However, despite the fact that failure detectors are unreliable, they are still useful for achieving distributed consensus [CT96, CHT96]. An atomic broadcast is a broadcast which guarantees that all participants in a system eventually receive the same sequence of messages. It is also a powerful primitive in distributed systems and was shown to be equivalent to distributed consensus [CT96].

Early solutions to consensus can be found in the systems such as Viewstamped Replication [OL88], Gbcast [Bir85, BJ87] and in the work of Dwork *et al.* [DLS88]. At the same time, state machine replication, introduced by Lamport [Lam78b] and popularised by Schneider [Sch90], emerged as a technique to make applications fault-tolerant by replicating the application state and coordinating its operations using consensus.

Eight years after its submission in 1990, the infamous Part-Time parliament paper [Lam98] describing Paxos was published, by which time attempts to explain the algorithm in simpler terms had already begun [PLL97] and continue today [Lam01a, Lam01b, VRA15]. Paxos became the de facto approach to distributed consensus and thus became the subject of extensive follow-up research, examples of particular relevance to this thesis include Disk Paxos [GL03], Cheap Paxos [LM04], Fast Paxos [Lam05a] and Egalitarian Paxos [MAK13]. The common foundation between Paxos and earlier proposed solutions to consensus has been noted elsewhere in the academic literature [Lam96, vRSS15, LC12].

In 2007, Google published a paper documenting their experience of deploying Paxos at scale [CGR07] in the Chubby locking service [Bur06]. Chubby was in turn used for distributed coordination and metadata storage by Google systems such as GFS [GGL03] and Bigtable [CDG+08]. This was shortly followed by the Zookeeper coordination service [JRS11, HKJR10], referred to by some as the open source implementation of Chubby. The project became very popular and is credited with bringing distributed consensus to the masses. Meanwhile, the idea of utilising Paxos for state machine replication was improving community understanding and adoption of distributed consensus [BBH+11, LC12, OO14].

The result has been a recent resurgence of distributed consensus in production today[2] to

[2]Implementations include Zookeeper (`zookeeper.apache.org`), Consul (`www.consul.io`) and Etcd (`coreos.com/etcd`).

provide key-value stores and coordination services[3].

1.3 Motivation

Despite becoming the de facto approach to consensus in distributed systems, Paxos is not without its limitations.

Firstly, Paxos is notoriously difficult to understand, leading to much follow up work, explaining the algorithm in simpler terms [PLL97, Lam01a, OO14, VRA15] and filling the gaps in the original description, necessary for constructing practical implementations [CGR07, BBH$^+$11]. This dissonance between the theory and the systems communities is best illustrated by the following quotes:

> The Paxos algorithm, when presented in plain English, is very simple.
>
> [Paxos] is among the simplest and most obvious of distributed algorithms.
>
> - Leslie Lamport [Lam01a]

> Paxos is exceptionally difficult to understand. The full explanation is notoriously opaque; few people succeed in understanding it, and only with great effort. . . . In an informal survey of attendees at NSDI 2012, we found few people who were comfortable with Paxos, even among seasoned researchers.
>
> We concluded that Paxos does not provide a good foundation either for system building or for education. - Diego Ongaro and John Ousterhout [OO14]

Secondly, the reliance on majority agreement means that the Paxos algorithm is slow to reach decisions, as each requires a round trip to/from many participants. By involving most participants in each decision, a high load is placed upon the network between participants and the leader itself. As a result, systems are limited in scale, often to three or five participants[4], as each additional participant substantially decreases overall performance[5]. It is widely understood that Paxos is unable to reach an agreement if the majority of participants have failed. However, this is only part of the overall picture, failure to reach agreement can result not only from unavailable hosts but also network partitions, slow hosts, network congestion, contention for resources such as persistent storage, clock skew, packet loss and countless other scenarios. Such issues are commonplace in some systems,

[3] Applications include databases such as HBase (`hbase.apache.org`) or MongoDB (`mongodb.com`) and orchestration tools such as Kubernetes (`kubernetes.io`), Docker Swarm (`github.com/docker/swarm`) and Mesos (`mesos.apache.org`)

[4] For example, Chubby reaches consensus between a small set of servers, typically five [CGR07]. Likewise, Raft clusters typically contain five servers [OO14, §5.1]

[5] This effect can be seen for example in [MJM08, Figure 8]

they are often correlated and escalated by one another. In practice, deploying Paxos does not guarantee availability since the algorithm's progress depends on satisfying synchrony and liveness conditions which cannot be guaranteed by today's systems.

Paxos's approach to consensus establishes one participant as the leader and makes that participant responsible for decisions. This centralised approach provides simplicity as a single point of serialisation yet it also bottlenecks the algorithm's performance to that of a single highly congested participant. Since the leader is responsible for decision making, all requests for decisions must be forwarded to and handled by the leader, further increasing decision latency. The leader introduces a single point of failure in the distributed system. Whilst Paxos is able to recover from leader failure under given conditions, such a recovery may be slow and cumbersome and usually results in a period of unavailability.

The limitations are widely known, yet few alternatives to Paxos are utilised in practice. The vast academic literature in distributed consensus generally focuses on mitigating these limitations though optimisation, extension and pragmatic implementation. Given the limitations we have discussed thus far, production systems such as Amazon's Dynamo [DHJ+07] and Facebook's TAO [BAC+13, LVA+15] opt to sacrifice strong consistency guarantees in favour of high availability.

1.4 Approach

The question naturally arises of whether these limitations are inherent to the problem of consensus or specific to the approach taken by the Paxos algorithm? Likewise, is the Paxos algorithm the optimal solution to consensus? These are the questions which will guide our research.

Our approach is to re-examine the problem of distributed consensus and how we as a community approach it. In contrast to previous work, we undertake an extensive examination of how to achieve consensus over a single value. Due to the wide spread adoption of Paxos and our focus on the underlying theory of consensus, the results of our analysis could have wide-reaching implications, which are agnostic (thus not limited in scope) to particular systems, hardware, workloads or deployment scenarios.

We begin by developing a framework for proving the correctness of a consensus algorithm and apply it to the Paxos algorithm. The purpose of the framework is to be explicit about how the properties of the algorithm are used within the proof of correctness. This allows us to modify the algorithm and verify correctness without re-proving the whole algorithm. The surprising results of this approach are twofold: firstly, the proof of correctness did not use the full strength of the properties provided and secondly, there are many approaches which satisfy the same properties. These observations formed the basis of our progressive generalisation of the Paxos algorithm. At each stage, we were able to verify correctness by building upon the original proof.

1.5 Contributions

This thesis is divided into 8 chapters, over which we construct novel generalised algorithms for solving distributed consensus by progressively generalising the popular Paxos algorithm. Overall, we make the following key contributions:

Chapter 2 We begin by defining the problem of distributed consensus and outline two known solutions, a simple straw-man algorithm and the widely used Paxos algorithm. We prove that both algorithms satisfy the necessary requirements to solve consensus.

Chapter 3 In this systematisation of knowledge chapter, we outline the most common refinements to the Paxos algorithm, separating the underlying algorithmic contributions from the particulars of the framing and terminology used in the literature, which often differs greatly between publications.

Chapter 4 We generalise the Paxos algorithm by weakening the quorum intersection requirements to permit non-intersecting quorums within each of the algorithm's two phases. We then propose a further generalisation by weakening the quorum intersection requirements to permit non-intersecting quorums between the algorithm's first phase and subsequent second phases.

Chapter 5 We prove that quorum intersection is transitive and can be reused, allowing in some scenarios decisions to be reached with fewer participants.

Chapter 6 We generalise the Paxos algorithm by weakening the value selection rules by utilising knowledge from the algorithm's first phase. This generalisation allows participants more flexibility when choosing a value to propose.

Chapter 7 We further extend our generalisation permitting various mechanisms for the sharing of phases to best take advantage of the generalisation thus far. We present algorithms which provide new progress guarantees and can reach decisions in few phases.

The result of this thesis is a family of approaches to achieving distributed consensus, which generalises over the most popular existing algorithms such as Paxos and Fast Paxos [Lam05a]. We aim to further understanding of this often poorly understood field and demonstrate the breadth of possible correct approaches to solving consensus. Later in the thesis, we explore the wide-reaching implications of our revised understanding of consensus. We focus on how to improve the performance and reliability of consensus algorithms and thus the distributed systems built on top of them. Distributed systems are famous for the need to compromise between desirable properties, largely due to popular formulations such as CAP theorem. However, such formulations are crude. In contrast, we aim to quantify the specific tradeoffs available for consensus and demonstrate algorithms which achieve these properties.

1.5.1 Publications

Parts of the research described in this thesis have been published in the following peer-reviewed conference and journal papers:

Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible Paxos: Quorum intersection revisited. In Proceedings of the 20th International Conference on Principles of Distributed Systems (OPODIS), 2016

The following publications are outside scope for inclusion:

Heidi Howard, Malte Schwarzkopf, Anil Madhavapeddy, and Jon Crowcroft. Raft Refloated: Do we have consensus? SIGOPS Operating Systems Review, 49(1):1221, January 2015

Amir Chaudhry, Jon Crowcroft, Heidi Howard, Anil Madhavapeddy, Richard Mortier, Hamed Haddadi, and Derek McAuley. Personal data: Thinking inside the box. In Proceedings of The Fifth Decennial Aarhus Conference on Critical Alternatives, AA 15, pages 2932. Aarhus University Press, 2015

1.5.2 Follow up research

This research in this thesis is by no means the final word on distributed consensus. In fact, it leaves many more doors open then it closes. Paxos revision A, which we will describe in §4.1, was published under the name *Flexible Paxos* and at the time of writing, follow up research and systems development by the community has already begun:

1. Formal specification of Flexible Paxos in Pluscal [Dem] and mechanised formal verification of Flexible Paxos using decidable logic [PLSS17].
2. Consensus protocols for graphically distributed systems using Flexible Paxos's weakened quorum intersection requirements such as WPaxos [ACDK17] and DPaxos [NAEA18]
3. Various implementation including Trex [Tre], a Flexible Paxos prototype for JVM and the adaption of Apache Zookeeper to use Flexible Paxos [Mel17].

1.6 Scope & limitations

Our approach has the following limitations:

Byzantine fault tolerance – We assume that algorithms are implemented and executed correctly. Participants and the network between them cannot act arbitrarily or maliciously. Consensus algorithms which do not assume this are known as *byzantine fault tolerant*. PBFT [CL99] is an example of such an algorithm.

Reconfiguration – We assume a fixed and known set of participants each with a unique identifier. Reconfiguration is discussed extensively in the literature and is a component of many algorithms. Examples include Stoppable Paxos [MLZ08], VRR [LC12, §7], Raft[OO14, §6].

Weakened semantics – We do not support operations with weakened semantics such as stale reads or operations which rely on synchrony or bounded clock drift for safety such as master leases [Bur06, VRA15].

Implementation details – We assume unbounded storage, representation of arbitrary values, no corruption to state or messages. Participants may stop and restart. Upon restarting, the persistent state is unchanged, non-persistent state is re-initialised and the algorithm is executed again from the beginning. The pseudocode provided in this thesis is assumed to be executed in order by a single thread and each line is executed atomically. Writes to state must be completed before proceeding, including writes to persistent storage. This can be achieved by techniques such as Write-ahead logging [MHL+92]. Reads from state must always return an up to date value.

Partial ordering – Our algorithms decide a single value (or decide a totally ordered, infinite sequence of values). We do not consider agreement over multiple series of values, partially ordered sequences [Lam05b] or finite sequences [MLZ08].

Progress in practice – Participants may operate at arbitrary speeds. Messages are eventually delivered but there is no bound on the time for the communication channel to deliver messages. Messages may be delivered out-of-order or multiple times. However, the progress of the algorithms depends on extensive assumptions, including synchrony and timing. We prove the progress for our algorithms under these assumptions however they are not minimal.

Specific systems – All algorithms are provided as high-level representations, not concrete protocols or implementations. To remain applicable to a range of existing and further systems, we do not optimise for particular systems or workloads as has been the subject of extensive research. For example, Ring Paxos[MPSP10] and Multi-Ring Paxos[MPP12] optimise for networks providing IP multicast.

Chapter 2

Consensus & Classic Paxos

We begin our study of distributed consensus by first considering how to decide upon a single value between a set of participants. This task, whilst seemingly simple, will occupy us for the majority of this thesis. Single-valued agreement is often overlooked in the literature as already solved or trivial and is seldom considered at length, despite being a vital component in distributed systems which is infamously poorly understood.

This chapter is broadly divided into three parts. We begin by defining the requirements for an algorithm to solve distributed consensus (§2.1). Secondly, we outline two existing algorithms which solve single valued consensus: the single acceptor algorithm (§2.1.1), a naïve straw man solution and Classic Paxos (§2.2, §2.3, §2.4), the widely adopted solution which lies at the foundation of a broad range of complex distributed systems. Thirdly and finally, we go on to prove that both algorithms satisfy all of the requirements of distributed consensus as defined in the first part (§2.5,2.6,2.7).

2.1 Preliminaries

Single valued distributed consensus is the problem of deciding a single value $v \in V$ between a finite set of n participants, $U = \{u_1, u_2, \ldots, u_n\}$.

Definition 1. *An algorithm is said to solve distributed consensus only if it satisfies the following three safety requirements:*

Non-triviality *The decided value must have been proposed by a participant.*

Safety *If a value has been decided, no other value will be decided.*

Safe learning *If a participant learns a value, it must learn the decided value.*

In addition to the following two progress requirements[1]*:*

Progress *Under a specified set of liveness conditions, if a value has been proposed by a participant then a value is eventually decided.*

Eventual learning *Under a specified set of liveness conditions, if a value has been decided then a value is eventually learned.*

Together, these five requirements prevent many trivial algorithms from satisfying distributed consensus. Without the safety and safe learning requirement, then a consensus algorithm could decide/learn all values proposed by participants. If non-triviality was not required, then a consensus algorithm could simply decide a fixed value. If progress and eventual learning were not required, then a consensus algorithm could never decide a value, rejecting all proposals it receives or never allowing any participant to learn the value. These trivial approaches are of little interest thus all five of the stated requirements are necessary.

It is important to note that the safety requirements do not rely on any liveness conditions. In other words, failures or asynchrony cannot lead to a violation of safety thus the algorithm cannot depend upon bounded clock drift, message delay, or execution time.

In contrast, progress can rely upon specified liveness conditions such as partial synchrony. The liveness conditions are always sufficient for the algorithm to make progress, regardless of the state of the system. In other words, the algorithm cannot become indefinitely stuck in deadlock (or livelock).

Notice that none of the requirements restrict which proposal value is decided. Specifically, an algorithm for distributed consensus is free to choose from any of the proposed values, regardless of which participant proposed the value, the proposal order, the number of participants proposing the same value or the proposed values themselves. The only restriction is that, under the progress condition, a value must eventually be decided if at least one value has been proposed. Therefore, from the non-triviality condition, we observe that if only one value is proposed then it must eventually be chosen.

In this chapter, we will formulate the problem of consensus in the usual manner adopted in the academic literature[2]. Each participant in the system is assigned one or both of the following two roles.

- Proposer - A participant who wishes to have a particular value chosen.

[1] Note, that this definition of progress is more general than the one commonly found in the literature which is specific to majorities. This generalisation aims to decouple majorities (a common aspect of consensus algorithms) from the problem definition.

[2] This approach is taken to aid readers who are already familiar with the field, although, it can be ambiguous at times.

- Acceptor - A participant who agrees and persists decided values.

In a system U of n participants, we will denote the set acceptors as $A = \{a_1, a_2, \dots\}$ where $A \subseteq U$ and $|A| = n_a$ and the set of proposers as $P = \{p_1, p_2, \dots\}$ where $P \subseteq U$ and $|P| = n_p$. A consensus algorithm defines the process by which a value v is chosen by the acceptors from the proposers. We refer to the point in time when the acceptors have committed to a particular value as the *commit point*. After this point in time, v has been decided and cannot be subsequently altered. The proposers learn which value has been decided, this must always take place after the commit point has been reached.

If we are able to reach agreement over a single value, we are able to reach agreement over an infinite sequence of values $v_1, v_2, v_3, \dots$ [3] by independently reaching consensus over each value in the sequence in turn. This sequence could represent updates to a re-writeable register, operations for a replicated state machine, messages for atomic broadcast, a shared log or state changes in a primary-backup system.

All notation introduced in this section and the remainder of the thesis is summarised for reference in Table 2.1.

2.1.1 Single acceptor algorithm

In the section, we introduce a straw-man algorithm which solves distributed consensus. The algorithm, which we will refer to as the *single acceptor algorithm (SAA)*, requires that exactly one participant be assigned the role of acceptor[4]. The liveness conditions for SAA are that the acceptor and at least one proposer are up and can exchange messages reliably. This algorithm is included here to familiarise the reader with the terminology and methodology before we progress to more advanced algorithms.

The single acceptor algorithm chooses the first value proposed by the proposers. A proposer who has a candidate value γ will propose the value to the acceptor using the message *propose(*γ*)*. If this is the first proposal the acceptor has received, it will write γ to persistent storage (known as *accepting*) and notify the proposer that the value has been decided using the message *accept(*γ*)*. Otherwise, if this is not the first proposal to be received, the acceptor will reply to the proposer with the already decided value γ' using *accept(*γ'*)*. In either case, provided that the acceptor is available then the proposer will learn the decided value. See algorithm 1 and algorithm 2 for pseudocode descriptions of this approach[5].

[3] The sequence of values will always be 1-indexed.

[4] We are not the first to use this straw-man solution to explain the problem of consensus [Lam01a, §2.2]

[5] Note that for all pseudocode in this thesis, variables are stored in volatile memory and are initially *nil* unless otherwise stated. Also note that all pseudocode in this thesis favours clarity and consistency over performance.

Notation	Description	First use:
$u_1, u_2, \ldots$	participants	2.1
$a_1, a_2, \ldots / p_1, p_2, \ldots$	specific acceptors/proposers	2.1
$n/n_a/n_p$	number of participants/acceptors/proposers	2.1
$v, w, x, \ldots$	values	2.1
$v_1, v_2, \ldots$	sequence of values	2.1
$a, a', \ldots / p, p', \ldots$	acceptors/proposers	2.1.1
$A, B, C \ldots$	concrete values	2.1.1
γ, γ'	candidate values	2.1.1
v_{acc}	last accepted value	2.1.1
$e, f, g, \ldots$	epochs	2.2
(e, v)	proposal with epoch e and value v	2.2
e_{min}/e_{max}	minimum/maximum epoch	2.2
e_{pro}/e_{acc}	last promised/accepted epoch	2.2
v_{dec}	decided value	3.3
$pid/sid/vid$	proposer ID/sequence ID/version ID	3.8
p_{lst}	last proposer	3.9
$U/A/P$	set of participants/acceptors/proposers	2.1
V	set of values	2.1
E	set of epochs	2.2
$\mathcal{E}$	set of unused epochs	2.2
Q_P/Q_A	set of acceptors which have promised/accepted	2.2
Q_V	set of acceptors which have promised with e_{max}	3.2
Γ	set of candidate values	2.2
$Q, Q', \ldots$	quorums (set of acceptors)	3.11
$\mathcal{Q}, \mathcal{Q}', \ldots$	quorum set	3.11
$\mathcal{Q}_i^e$	quorum set for phase i and epoch e	4.1
V_{dec}	set of values which maybe decided	6.1
R	mapping from acceptors to a promise, *no* or (e, v)	6.1
D	mapping from quorums to decisions	6.1
$min(\mathcal{E})$	returns minimum epoch in $\mathcal{E}$	2.2
$succ(e)$	returns successor of epoch e	3.8
$only(V)$	returns the only element in singleton set V	6.1

Table 2.1: Reference table of notation.

Algorithm 1: Proposer algorithm for SAA

```
    state:
      • γ: candidate value (configured, persistent)

1   send propose(γ) to acceptor
2   case accept(v) received from acceptor
      /* proposer learns that v was decided so return v */
3     return v
```

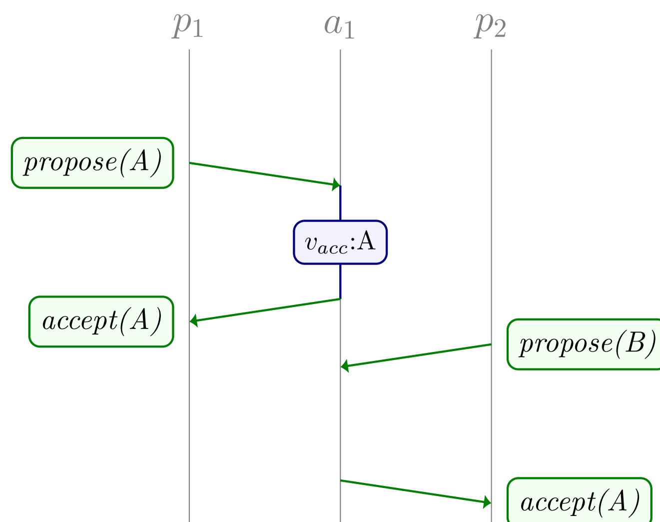

Figure 2.1: Example run of SAA between one acceptor $\{a_1\}$ and two proposers $\{p_1, p_2\}$.

Algorithm 2: Acceptor algorithm for SAA

```
    state:
      • v_acc: accepted value (persistent)

1   while true do
2     case propose(v) received from proposer
3       if v_acc = nil then
4         v_acc ← v
5       send accept(v_acc) to proposer
```

Figure 2.1 is a message sequence diagram (MSD) for an example execution of the single acceptor algorithm. We will make extensive use of MSDs to illustrate the messages exchanged and state updates which occur over time. Note that the time axis (the negative y-axis) is not assumed to be linear. In this example, proposer p_1 has candidate value $\gamma = A$ and proposer p_2 has candidate value $\gamma = B$. The acceptor receives *propose(A)* first and therefore the value A is decided.

Safety

Informally, we can see that this simple algorithm satisfies the three safety conditions for distributed consensus. The acceptor chooses the first proposal it receives thus it satisfies non-triviality. After accepting its first proposal, the acceptor accepts no other proposals from proposers thus this algorithm satisfies safety. If a proposer returns a value, then the value must have been received from the acceptor and therefore must be decided, satisfying safe learning. We will now consider these in more detail.

Theorem 1 (Non-triviality of SAA)**.** *If the value v is decided, then v must have been proposed by a proposer*

Proof of theorem 1. Assume the value v is decided. For v to be decided, the acceptor must have accepted the proposal *propose(v)*. Thus since messages cannot be corrupted, v must have been proposed by some proposer □

Theorem 2 (Safety & safe learning of SAA)**.** *For any two proposers $p, p' \in P$, which learn that the decided value v is γ and γ' respectively then $\gamma = \gamma'$.*

Proof of theorem 2. A proposer p learns that the decided value v is γ as a result of receiving *accept(γ)* from the single acceptor. The same is true for any other participant p'.

Since the events of sending *accept(γ)* and sending *accept(γ')* occur on one participant, the single acceptor, the two events cannot have occurred concurrently. Thus one event must happen before the other.

Assume that the event send *accept(γ)* is before send *accept(γ')*.

The acceptor determines the values γ, γ' by reading the accepted value v_{acc}. If $\gamma \neq \gamma'$ then the value v_{acc} would have changed from γ to γ' between sending the two accepted messages. The only mechanism for the accepter to update v_{acc} to γ' is by receiving *accept(γ')*. Updating v_{acc} is conditional on v_{acc} being nil, since v_{acc} is persistent, it cannot be nil as it has previously been set to γ. Therefore v_{acc} cannot have been updated between the two sending events so $\gamma = \gamma'$.

The same applies when send *accept(γ')* is before send *accept(γ)*. □

Progress

Informally, we can see that this simple algorithm also satisfies the two progress conditions for distributed consensus, under the liveness conditions that the acceptor and at least one proposer must be up. Note, that though we use the assumption that messages are eventually delivered, we do not require time bounds on message delivery or operating speed.

Theorem 3 (Progress of SAA)**.** *If a proposer $p \in P$ proposes the value γ and the liveness conditions are satisfied for a sufficient period then a value v is eventually decided.*

Proof of theorem 2. Assume proposer p sends *propose(γ)* to the acceptor. Under the liveness conditions, this message will be eventually received by the acceptor. Under the liveness conditions, this acceptor must be up and handle the message. Either no decision has yet been reached thus the proposal is accepted and $v = \gamma$, otherwise a decision has already been reached and $v = v_{acc}$. □

Summary

This simple algorithm provides consensus in one round trip (two messages) to the acceptor and one synchronous write to persistent storage, provided the acceptor is up. If the acceptor is down, then the system cannot progress until the acceptor is up. This algorithm works as all value proposals intersect at a single point, the acceptor. The result is that proposals are totally ordered so choosing a proposal is trivial. However, this reliance on a single acceptor is also this algorithm's downfall. Should this acceptor fail, the algorithm could not make progress until it recovers.

The acceptor in SAA is a single point of failure, the obvious step to address this is to have multiple acceptors. However, we can no longer guarantee a total ordering over proposals so the single acceptor algorithm is no longer suitable[6]. In the next section, we instead describe Classic Paxos, a consensus algorithm which is able to handle multiple acceptors.

2.2 Classic Paxos

Classic Paxos [Lam98][7] is an algorithm[8] for solving the problem of distributed consensus. In the best case, the unoptimised algorithm is able to reach agreement in two round trips to the majority of acceptors and three synchronous writes to persistent storage, though in some cases more time will be needed. The liveness conditions is that $\lfloor n_a/2 \rfloor + 1$ of n_a acceptors and one proposer must be up and communicating synchronously. These conditions are both necessary and sufficient for progress.

The approach taken by Classic Paxos to deciding a value has two phases. Phase one can be viewed as the reading phase, where the proposer learns about the current state of the system and takes a type of version number to detect changes in the future. Phase two can be viewed as the writing phase, where the proposer tries to get a value accepted. If, after phase one of the algorithm, the proposer is certain that a value has not yet been decided,

[6] Assuming the network does not provide atomic broadcast.

[7] Also known as *Synod* or *Single-degree Paxos*

[8] More correctly, it is a family of algorithms

the proposer can propose the candidate value γ. If the outcome of phase one is that a value might already be decided, then that value must be proposed in phase two instead. Each of these two phases requires a majority of acceptors to agree in order to proceed.

We now define the terms *epoch* and *proposal* and then use these to summarise the Classic Paxos algorithm.

Definition 2. *An epoch e is any member of the set of epochs E. E is any infinite totally ordered set such that the operators $<$, $>$ and $=$ are always defined* [9].

Definition 3. *A proposal (e, v) is any epoch and value pair*[10].

Classic Paxos Phase 1

1. A proposer chooses a unique epoch e and sends *prepare(e)* to the acceptors.
2. Each acceptor stores the last promised epoch and last accepted proposal. When an acceptor receives *prepare(e)*, if e is the first epoch promised or if e is equal to or greater than the last epoch promised, then e is written to storage and the acceptor replies with *promise(e,f,v)*. *(f,v)* is the last accepted proposal (if present) where f is the epoch and v is the corresponding proposed value.
3. Once the proposer receives *promise(e,_,_)* from the majority of acceptors, it proceeds to phase two. Promises may include a last accepted proposal which will be used by the next phase.
4. Otherwise if the proposer times out, it will retry with a greater epoch.

Classic Paxos Phase 2

1. The proposer must now select a value v using the following *value selection rules*:

   i If no proposals were returned with promises in phase one, then the proposer will choose its candidate value γ.

   ii If one proposal was returned, then its value is chosen.

   iii If more than one proposal was returned then the proposer must choose the value associated with the greatest epoch.

   The proposer then sends *propose(e,v)* to the acceptors.

[9] Epochs are also referred to as terms [OO14, §5.1], view numbers [LC12, §3], round numbers [MPSP10, §3], instance values/epoch [HKJR10, §1] or ballot numbers in the literature

[10] Proposals are also referred to as ballots in the literature

2. Each acceptor receives a *propose(e,v)*. If e is the first epoch promised or if e is equal to or greater than the last promised epoch, then the promised epoch and accepted proposal is updated and the acceptor replies with *accept(e)*.

3. Once the proposer receives *accept(e)* from the majority of acceptors, it learns that the value v is decided.

4. Otherwise if the proposer times out, it will retry phase 1 with a greater epoch.

	Message	**Description**	**Sent by:**	**Received by:**
Phase 1	*prepare(e)*	e: epoch	proposers	acceptors
	promise(e,f,v)	e: epoch f: last accepted epoch* v: last accepted value* *maybe *nil*	acceptors	proposers
Phase 2	*propose(e,v)*	e: epoch v: proposal value	proposers	acceptors
	accept(e)	e: epoch	acceptors	proposers

Table 2.2: Messages exchanged in Classic Paxos

For reference, Table 2.2 gives an overview of the four messages used in Classic Paxos[11].

We will now look at this process in more detail.

2.2.1 Proposer algorithm

Algorithm 3 describes the Classic Paxos algorithm for participants with the role of a proposer. The key input to this algorithm is a candidate value γ to propose and the output is the decided value v. The decided value may or may not be the same as the candidate value, depending upon the state of acceptors when the algorithm is executed. The proposer will only propose its candidate value γ if it is sure that another value has not already been chosen. Once the proposer learns that a value has been decided, no proposer will learn that a different value has been decided.

After initialising its variables (Algorithm 3, lines 1-2), the algorithm begins by selecting a epoch e to use (Algorithm 3, line 3). To remain general, we do not specify how the set of available epochs, $\mathcal{E} \subseteq E$, should be generated. However, the algorithm does require that each proposer is configured with an infinite disjoint set of epochs. The algorithm ensures that each epoch is used only once, by removing the current epoch, e, from the set

[11]These message are often referred to as 1a, 1b, 2a and 2b respectively. Confusingly, the propose message is called prepare in VRR [LC12, §4.1]

Algorithm 3: Proposer algorithm for Classic Paxos

state:

- n_a: total number of acceptors (configured, persistent)
- e: current epoch
- v: current proposal value
- e_{max}: maximum epoch received in phase 1
- $\mathcal{E}$: set of unused epochs (configured, persistent)
- Q_P: set of acceptors who have promised
- Q_A: set of acceptors who have accepted

```
   /* (Re)set variables */
1  v, e_max ← nil
2  Q_P, Q_A ← ∅
   /* Select and set the epoch e */
3  e ← min(E)
4  E ← E \ {e}
   /* Start Phase 1 for epoch e */
5  send prepare(e) to acceptors
6  while |Q_P| < ⌊n_a/2⌋ + 1 do
7      switch do
8          case promise(e,f,w) received from acceptor a
9              Q_P ← Q_P ∪ {a}
10             if f ≠ nil ∧ (e_max = nil ∨ f > e_max) then
                   /* (e_max, v) is the greatest proposal received */
11                 e_max ← f, v ← w
12         case timeout
13             goto line 1
14 if v = nil then
       /* no proposals were received thus propose γ */
15     v ← γ
   /* Start Phase 2 for proposal (e,v) */
16 send propose(e,v) to acceptors
17 while |Q_A| < ⌊n_a/2⌋ + 1 do
18     switch do
19         case accept(e) received from acceptor a
20             Q_A ← Q_A ∪ {a}
21         case timeout
22             goto line 1
23 return v
```

of available epochs (Algorithm 3, line 4). For simplicity, we have the proposers try epochs in-order, though it is safe for the proposer to use any unused epoch.

The message *prepare(e)* is sent to all acceptors (Algorithm 3, line 5) and the proposer waits for responses. As promises are received, the proposer tracks the maximum epoch, e_{max}, received in a proposal and its associated value, v (Algorithm 3, lines 8-11). If a promise does not include a proposal then the maximum epoch, e_{max}, and its associated value, v, are not updated (Algorithm 3, line 10). The set $\mathcal{Q}_P$ tracks which acceptors have promised thus far. If promises are not received from a majority of acceptors before a timeout then the algorithm retries (Algorithm 3, lines 6, 12-13). If no proposals were received with promises, the proposal value v is set to the candidate value γ (Algorithm 3, lines 14-15).

The proposer then sends *propose(e,v)* to the acceptors (Algorithm 3, line 16). The proposer will return value v (Algorithm 3, line 23) after the majority of acceptors accept the proposal (e, v) (Algorithm 3, lines 17-20) and retry otherwise (Algorithm 3, lines 21-22).

Note that all other messages which are received by the proposer but do not match a switch statement (such as messages from previous epochs or promises during phase two) can be safely ignored.

2.2.2 Acceptor algorithm

Algorithm 4: Acceptor algorithm for Classic Paxos

state:
- e_{pro}: last promised epoch (persistent)
- e_{acc}: last accepted epoch (persistent)

```
1  while true do
2      switch do
3          case prepare(e) received from proposer
4              if e_pro = nil ∨ e ≥ e_pro then
5                  e_pro ← e
6                  send promise(e,e_acc,v_acc) to proposer
7          case propose(e,v) received from proposer
8              if e_pro = nil ∨ e ≥ e_pro then
9                  e_pro ← e
10                 v_acc ← v, e_acc ← e
11                 send accept(e) to proposer
```

The acceptors in Classic Paxos are responsible for handling incoming *prepare* and *propose* messages. The logic for this is described in Algorithm 4[12]. All messages, whether *prepare(e)*

[12]Algorithm 4 uses the variable v_{acc}, however it is not included in the state list. Due to space limitations,

or *propose(e,v)*, must have an epoch e greater than or equal to e_{pro} to be processed by the acceptor (Algorithm 4, lines 4,8). If this is the first message the acceptor has received then e_{pro} is *nil* and this test is always successful. If the test is successful then e_{pro} is updated to e (Algorithm 4, lines 5,9).

If the message was *prepare(e)*, then the acceptor replies with *promise(e,e_{acc},v_{acc})* (Algorithm 4, line 6). If the acceptor has not yet accepted a proposal then e_{acc} and v_{acc} will be *nil*. When an acceptor sends a promise message, we say that the acceptor has *promised* epoch e[13].

If the message was *propose(e,v)* then the acceptor will set e_{acc} and v_{acc} to the proposal (e, v) (Algorithm 4, line 10) and reply with *accept(e)* (Algorithm 4, lines 11). In this case, we say that the acceptor has *accepted* the proposal (e, v).

Definition 4. *In Classic Paxos, a proposal* (e, v) *is decided if the proposal* (e, v) *has been accepted by the majority of acceptors.*

Note that this definition does not require that the proposal is still the last accepted proposal on a majority of acceptors. A value $v \in V$ is said to be decided if there exists an epoch $e \in E$ such the proposal (e, v) has been decided. This is also described as value v is *decided* in e. The *commit point* is the first time a proposal is decided.

2.3 Examples

In this section, we will consider example message sequence diagrams (MSDs) for a sample of possible executions of Classic Paxos. For simplicity, messages are omitted if their receipt will have no effect. Each example system is comprised of three acceptors $A = \{a_1, a_2, a_3\}$ and two proposers $P = \{p_1, p_2\}$, thus $\lfloor n_a/2 \rfloor + 1 = 2$. Initially, $\gamma = A$ for proposer p_1 and $\gamma = B$ for proposer p_2.

In our examples, epochs are natural numbers $E = \mathbb{N}^0$, which have been divided round robin between the proposers. Therefore initially $\mathcal{E} = \{0, 2, 4, \dots\}$ on p_1 and $\mathcal{E} = \{1, 3, 5, \dots\}$ on p_2.

Figure 2.2 gives an example of two proposers executing Classic Paxos in serial. Firstly, proposer p_1 executes Classic Paxos and the proposal $(0, A)$ is decided. Then proposer p_2 executes Classic Paxos and the proposal $(1, A)$ is decided. Both proposers are able to complete Classic Paxos in two phases. This represents the best case scenario for Classic Paxos.

the state list for each algorithm only includes new variables. The descriptions of variables such as v_{acc} can be found in Table 2.1.

[13]The term *adopts* is sometimes used in the literature instead of promised, for example in [VRA15]

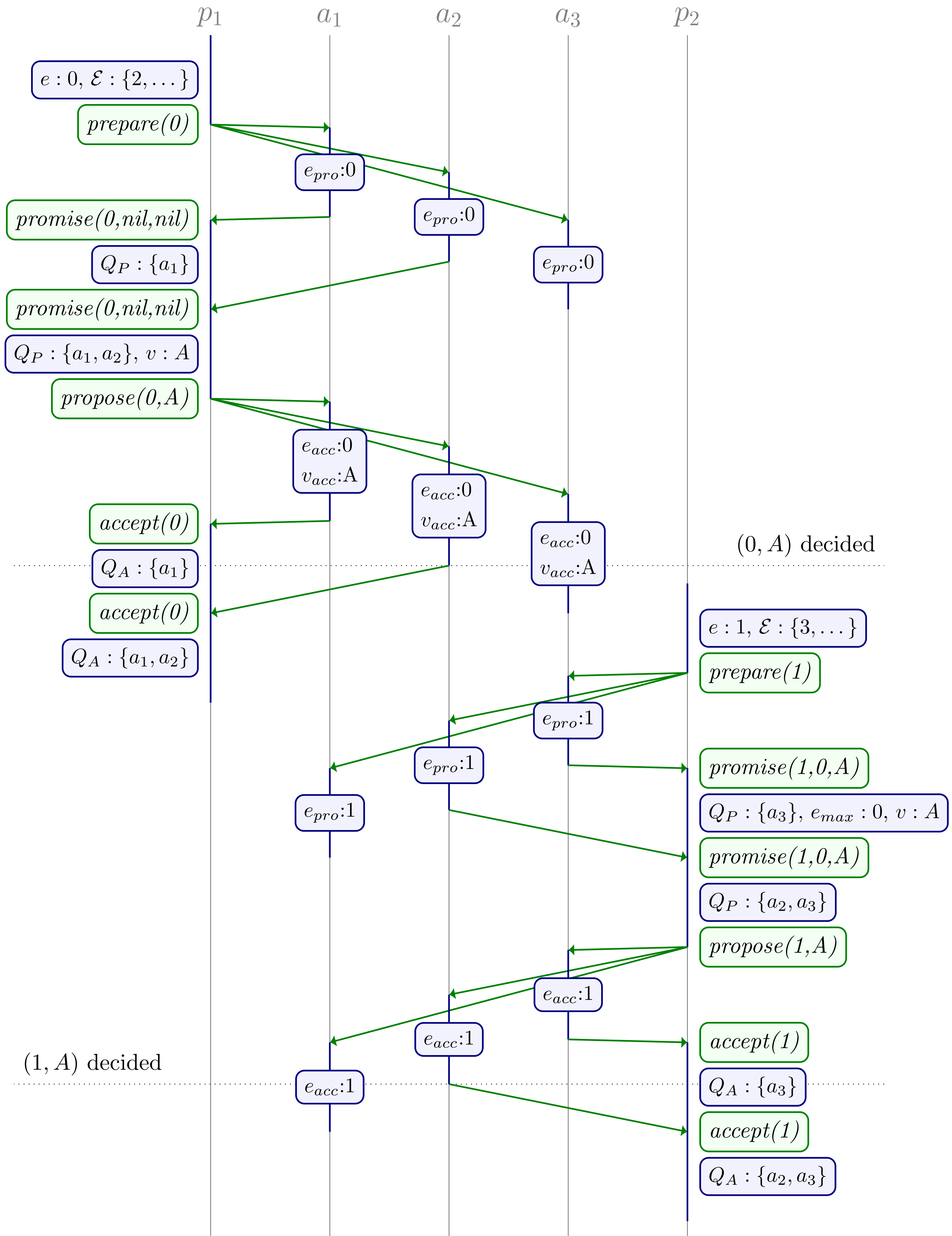

Figure 2.2: Example run of Classic Paxos with two serial proposers. Proposer p_1 executes Classic Paxos followed by the proposer p_2.

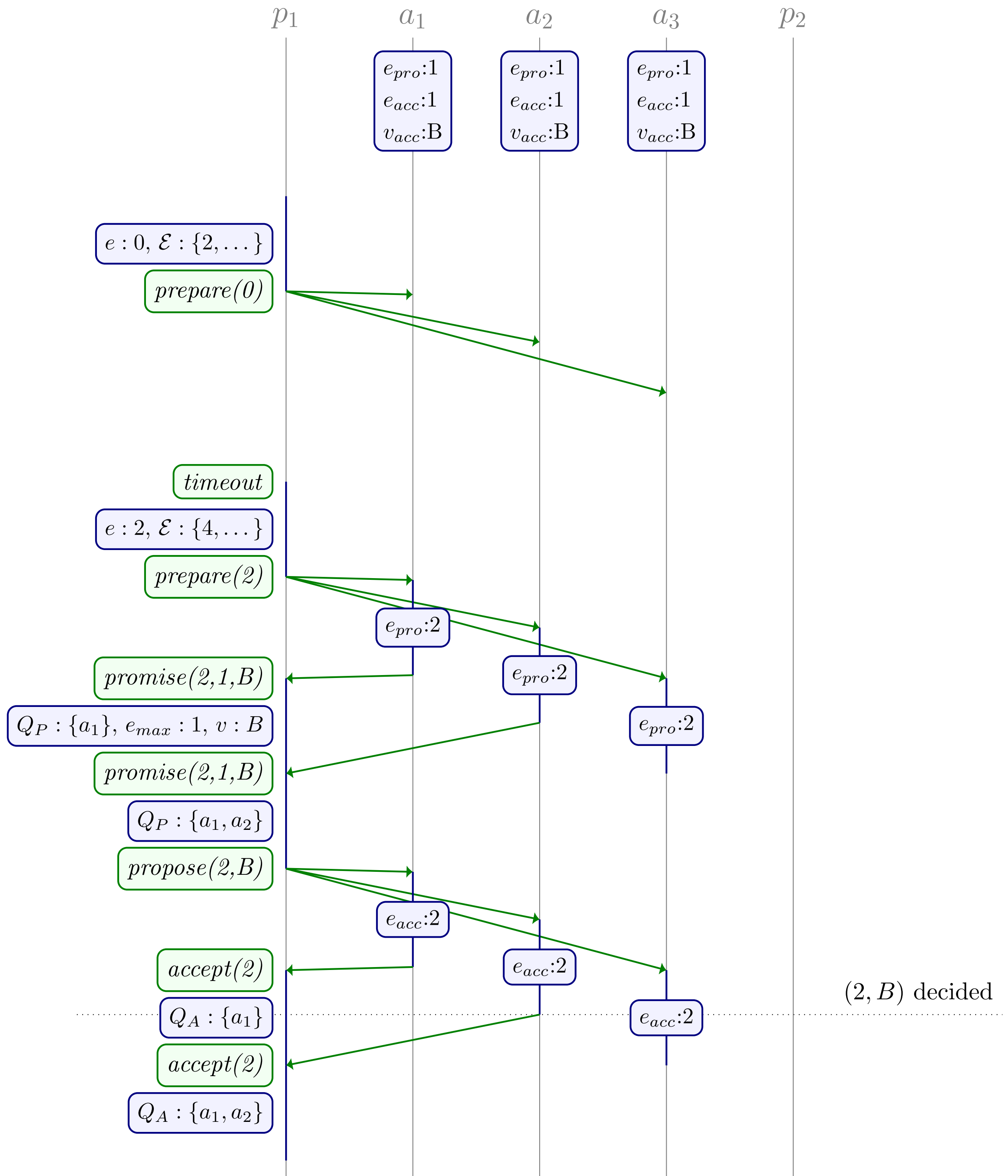

Figure 2.3: Example run of Classic Paxos with two serial proposers. Proposer p_2 has finished executing Classic Paxos before proposer p_1 begins.

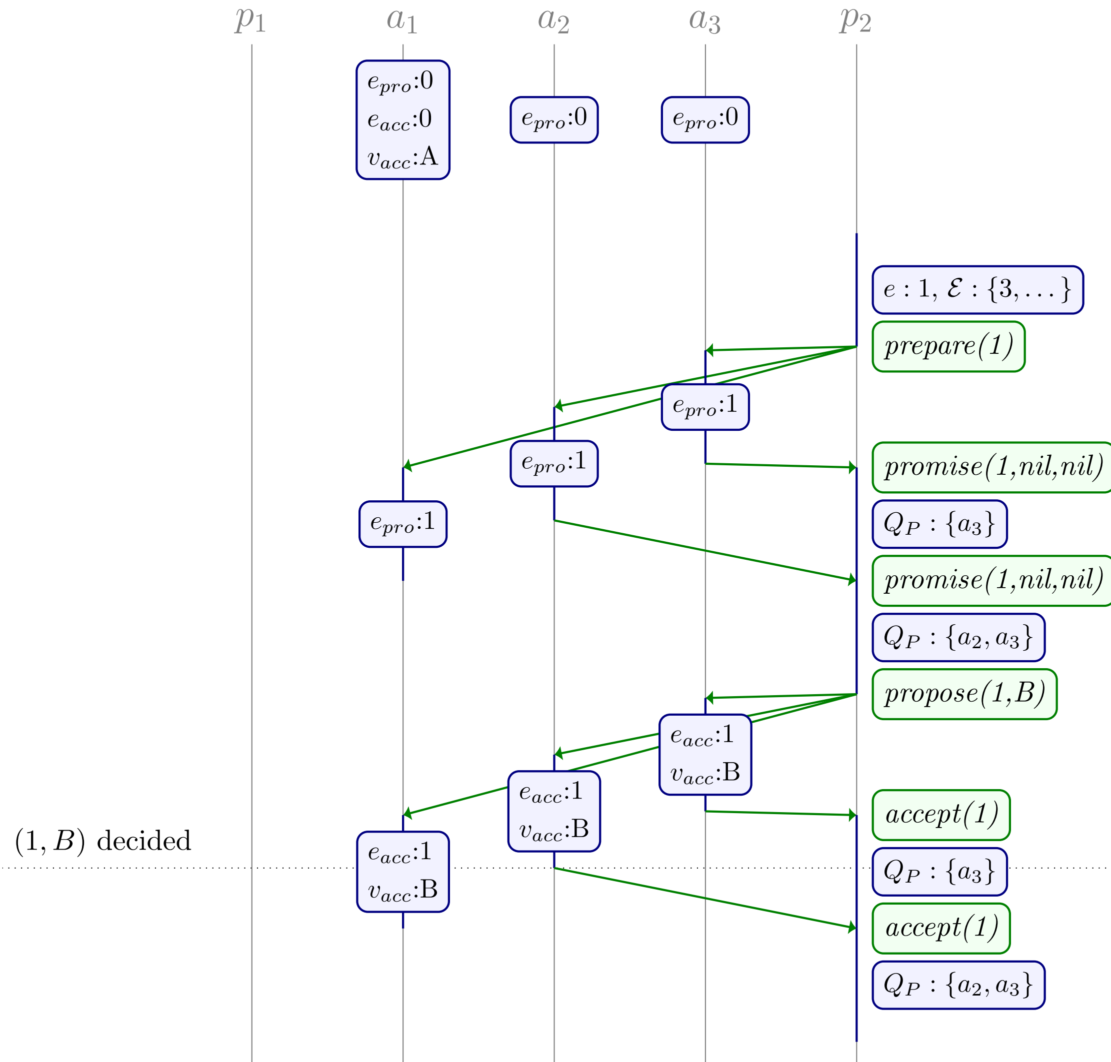

Figure 2.4: Example run of Classic Paxos where proposer p_1 stops during phase two prior to reaching the commit point. Proposer p_2 does not observe the proposal from p_1.

Initially, in Figure 2.3, proposer p_2 has executed Classic Paxos and the proposal $(1, B)$ has been decided and accepted by all acceptors. Subsequently the proposer p_1 executes phase one for epoch 0, however this phase is unsuccessful. The proposer p_1 retries Classic Paxos and the proposal $(2, B)$ is decided. Unlike before, proposer p_1 in this example required three phases to learn the decided value.

Figures 2.4 and 2.5 illustrate two possible outcomes if a proposer (in this case p_1) stops after making a proposal (in this case $(0, A)$) but prior to reaching the commit point. In Figure 2.4, proposer p_2 does not observe the proposal $(0, A)$ during its phase one thus the proposal $(1, B)$ is subsequently decided. In contrast, in Figure 2.5 the proposer p_2 does observe the proposal $(0, A)$ during its phase one thus the proposal $(1, A)$ is subsequently decided.

The examples thus far have demonstrated proposers executing Classic Paxos in serial. In Figure 2.6, we observe the worst case scenario of Classic Paxos when concurrent proposers

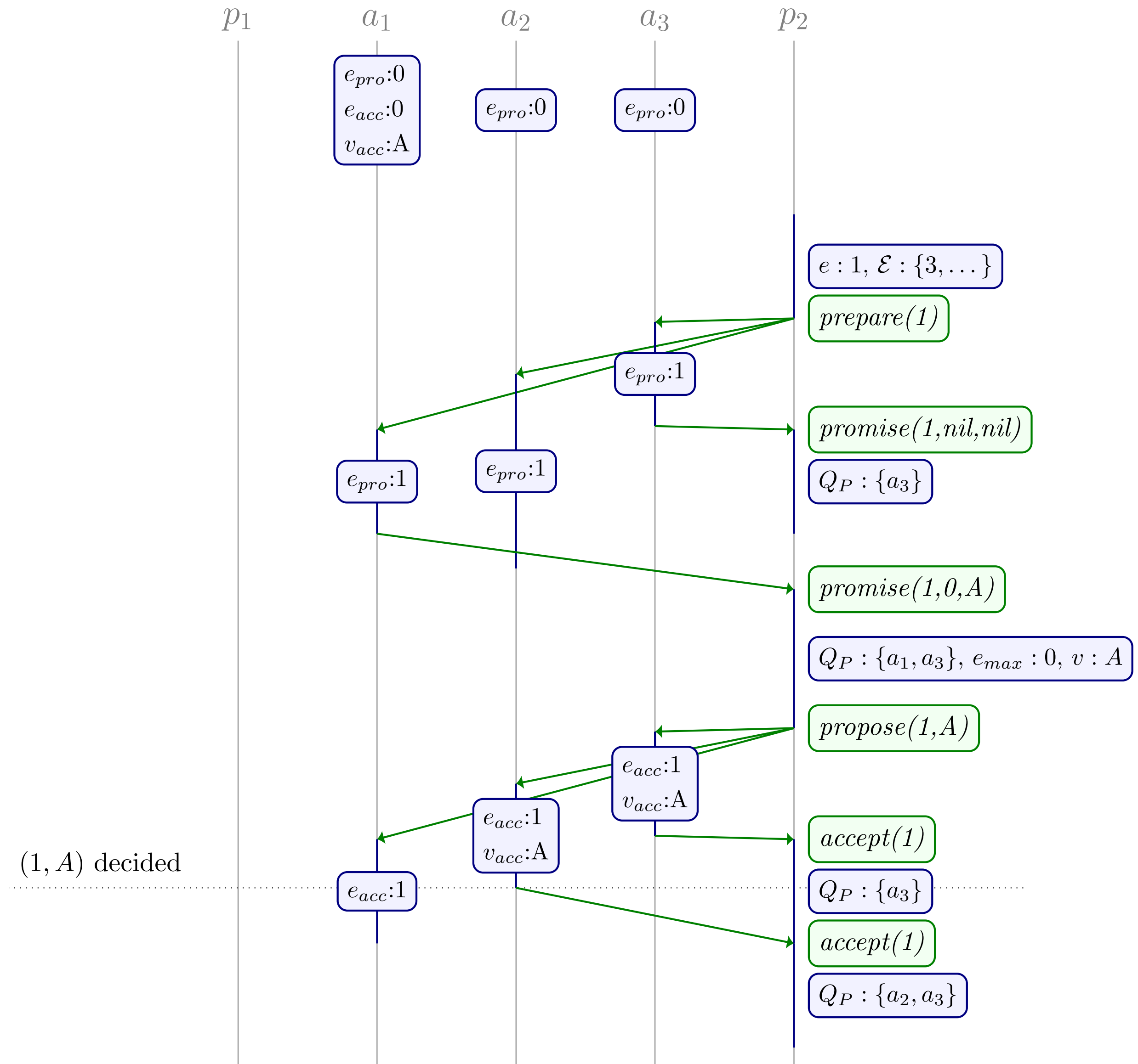

Figure 2.5: Example run of Classic Paxos where proposer p_1 stops during phase two prior to reaching the commit point. Proposer p_2 does observe the proposal from p_1.

duel such that neither proposer is able to make progress. Proposer p_1 executes phase one for epoch 0 then proposer p_2 executes phase one for epoch 1. Proposer p_1 is unsuccessful at phase two for proposal $(0, A)$ thus executes phase one for epoch 2. Proposer p_2 is then unsuccessful at phase two for proposal $(1, B)$. Though unlikely, this situation could continue indefinitely. Note that this situation can still occur when both proposers are proposing the same value or after a decision has been reached.

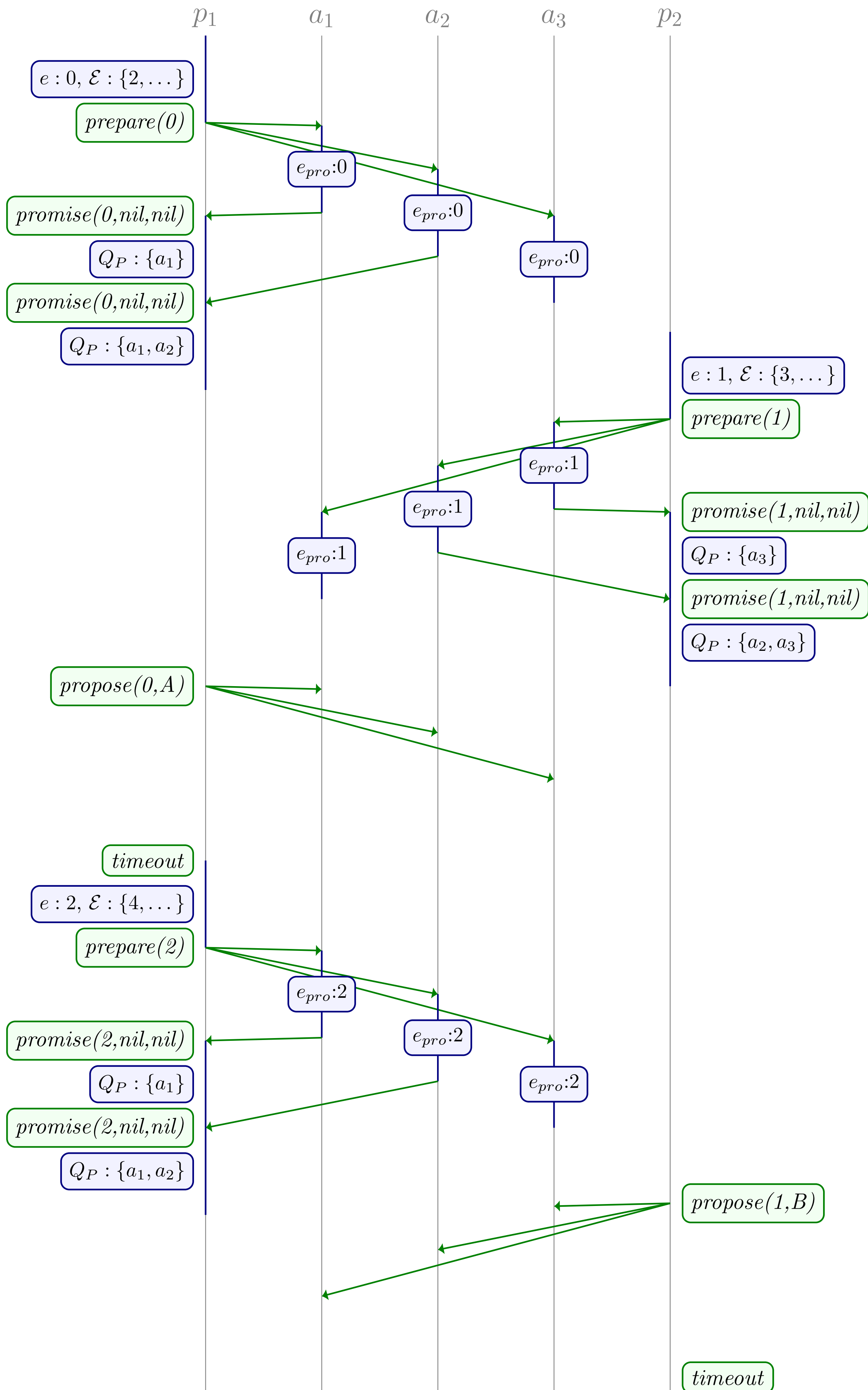

Figure 2.6: Example run of Classic Paxos with two concurrent proposers duelling.

2.4 Properties

Before we reason about the safety and liveness of Classic Paxos, we divide the algorithm into a set of properties. These properties will identify how specific components of the algorithm are utilised in subsequent proofs. In later chapters, we will modify the Classic Paxos algorithm, using these properties we will be able to determine which proofs are still valid and which need revising.

The key properties of the Classic Paxos proposer algorithm are as follows:

Property 1. *Proposers use unique epochs for each proposal.*

Property 2. *Proposers only propose a value after receiving promises from* $\lfloor n_a/2 \rfloor + 1$ *acceptors.*

Property 3. *Proposers only return a value after receiving accepts from* $\lfloor n_a/2 \rfloor + 1$ *acceptors.*

Property 4. *Proposers must choose a value to propose according to the value selection rules. If no previously accepted proposals were returned with promises then any value can be chosen. If one or more previously accepted proposals were returned then the value associated with the highest epoch is chosen.*

Property 5. *Each epoch used by a proposer is greater than all previous epochs used by the proposer.*

The key properties of the acceptor algorithm are:

Property 6. *For each* prepare *or* propose *message received by an acceptor, the message is processed by the acceptor only if epoch received is greater than or equal to the last promised epoch.*

Property 7. *For each* prepare *or* propose *message received, the acceptor's last promised epoch is set to the epoch received. This is after Property 6 has been satisfied.*

Property 8. *For each* prepare *message received, the acceptor replies with* promise*. This is after Properties 6 & 7 have been satisfied.*

Property 9. *For each* propose *message received, the acceptor replies with* accept *after updating its last accepted proposal. This is after Properties 6 & 7 have been satisfied.*

Property 10. *Last promised epoch and last accepted proposal are persistent and only updated by Properties 7 & 9*

In the following three sections (§2.5, §2.6 & §2.7), we prove that the Classic Paxos algorithm satisfies the requirements of non-triviality, safety and progress and thus is a solution to distributed consensus.

2.5 Non-triviality

Firstly, for Classic Paxos to solve distributed consensus it must satisfy non-triviality. Let Γ denote the set of candidate values proposed by proposers, thus non-triviality is specified as follows:

Theorem 4 (Non-triviality of decided values)**.** *If the value v is decided then $v \in \Gamma$.*

In Classic Paxos, for a value to be decided, it is necessary for the value to first be proposed. Therefore a stronger version of theorem 4 is:

Theorem 5 (Non-triviality of proposed values)**.** *If the value v is proposed then $v \in \Gamma$.*

Proof of theorem 5. Consider a proposer in phase two which proposes the value v with epoch e. We will let V denote the set of values which have been proposed so far, thus initially $V = \emptyset$.

We show by induction over the set of proposed values V that all proposed values are candidate values, thus $V \subseteq \Gamma$

Base case (initial state): Initially, before any values are proposed, $V = \emptyset$ and $\emptyset \subseteq \Gamma$.

Base case (the first proposal): Consider the first proposer to propose a value. We will denote this value as v. The value v must have been chosen according to the algorithm's value selection rules. Since no values have yet been proposed, no proposals will be received with the promises in the proposer's phase one. The first proposer will therefore always propose its own candidate value, $v \in \Gamma$ thus $V = v$ and $V \subseteq \Gamma$ (Property 4).

Inductive case: We assume that $V \subseteq \Gamma$ and that the next proposer proposes the value w. We will show that $w \in \Gamma$ thus $V \subseteq \Gamma$ remains true.

The value w must have been chosen according to the algorithm's value selection rules. It is either the case that no proposals were received during the proposer's phase one so the proposer proposed its own candidate value, $w \in \Gamma$; or one (or more) proposals were received with the promises in the proposer's phase one. The proposer will therefore propose the value w associated with highest epoch returned (Property 4). All proposals received must have been first proposed by a proposer. Therefore V remains unchanged and $V \subseteq \Gamma$ remains true. □

2.6 Safety

In order for the Classic Paxos algorithm to solve distributed consensus we must show that all possible executions of the algorithm are safe. In other words, if a value has been decided then no other value can also be decided. In this section, we will prove this property

of Classic Paxos, but first we begin by proving some simple properties of the algorithm, which will be of use to us later on.

Lemma 6 (Monotonicity of promises)**.** *The last promised epoch stored by each acceptor is monotonically increasing.*

Proof of lemma 6. The last promised epoch is initially nil and can only be updated by acceptors in response to receiving prepare or propose from proposers (Property 10). The last promised epoch is only updated to the epoch received if the epoch received is greater than or equal to last promised epoch (Properties 6 & 7).

Therefore the last promised epoch is strictly increasing. □

Lemma 7 (Relation between acceptor epochs)**.** *The last promised epoch is always greater than (or equivalent to) the last accepted epoch on each acceptor.*

Proof of lemma 7. Whenever the last accepted proposal is updated, the last promised epoch has always been updated to the same value (Properties 9 & 10). As a result, the last accepted proposal is never updated to a value strictly greater than the last promised epoch. Lemma 6 shows that the last promised epoch is monotonically increasing thus the last promised epoch is never updated to a value less than the last accepted epoch. Therefore, it is always the case that the last promised epoch $\geq$ last accepted epoch. □

The proof of lemma 7 highlights the importance of ensuring the steps in the Classic Paxos algorithm are executed in-order. If the last accepted proposal was written prior to writing the last promised epoch then an acceptor failure between these two writes could violate lemma 7.

Lemma 8 (General promise format)**.** *For all promises sent by acceptors of the form* promise(e,f,v)*, where $f \neq nil$ then it is the case that $e \geq f$.*

Proof of lemma 8. An acceptor would send *promise(e,f,v)* in response to receiving *prepare(e)* from a proposer (Property 8). Therefore $e \geq$ the last promised epoch when the prepare message was received. From lemma 7 the last promised epoch $\geq$ the last accepted epoch f. By transitivity on the $\geq$ relation, $e \geq f$ □

One implication of lemma 8 is that acceptors may send promises of the form *promise(e,e,v)*. This might occur if an acceptor was to receive a proposer's *propose(e,v)* before *prepare(e)* due to out-of-order delivery. However, a promise of this form will never be used by a proposer to complete phase one. This is because the proposer of e will have already completed phase one since (e, v) had already been proposed.

Corollary 8.1 (Useful promise format)**.** *All promises that are used by proposers towards a decision are either of the forms* promise(e,nil,nil) *(without a proposal) or* promise(e,f,v) *where $e > f$ (with a previous proposal).*

From this, we know that the greatest promise a proposer in e could receive is from the predecessor epoch[14]. Therefore according to the value selection rules (Property 4):

Corollary 8.2 (Predecessor proposals). *If a proposer in e receives* promise(e,f,v) *where* $e = succ(f)$ *during its phase one then the proposer will propose value* v.

Lemma 9 (Value uniqueness). *If the value v is proposed in epoch e then no other value can also be proposed in e.*

Proof of lemma 9. At most one proposer is able to use each epoch (Property 1). Each proposer will decide a value to propose and send a propose message to the acceptors with that value. A proposer will not use the same epoch twice. If a proposer fails during the proposal and does not have knowledge of the value chosen then it will start again with a new epoch. □

As a consequence of lemma 9:

Corollary 9.1 (Value uniqueness in promises). *For any two promises,* promise(_,f,v) *and* promise(_,g,w), *if* $f = g$ *then* $v = w$.

As a result, we know that a proposer will not receive multiple proposals in its phase one which have the same epoch but different values.

Lemma 10 (Message ordering). *If a series of messages has been sent by an acceptor*[15]*, then the message epochs are a partial ordering on the order in which the messages were sent. This applies regardless of whether the messages are all promises, all accepts or a combination of both.*

Proof of lemma 10. Consider two messages which have been sent by a acceptor with epochs e and f such that $e < f$. Assume that the message with epoch f was sent first.

When the acceptor sent the first messages, the last promised epoch, e_{pro}, will have been set to f, regardless of whether the message was promise or accept (Property 7). Lemma 6 shows that the last promised epoch is monotonically increasing so henceforth $e_{pro} \geq f$.

The second message has epoch e and is sent by the acceptor in response to a prepare or propose request, provided that $e \geq e_{pro}$ (Properties 6,8 & 9). This requires $e = f$, contradicting the assumption that $e < f$. Therefore the message with epoch f must be sent after the message with epoch e. □

Lemma 11 (Quorum intersection). *If a value v is decided in epoch e then at least one acceptor which accepted proposal* (e, v) *will be required to promise in any future proposals* $> e$.

[14]Note that when $e = min(E)$ no such predecessor exists.
[15]Whilst we do not prove it here, message ordering also applies to proposers.

Proof of lemma 11. Both phases of Classic Paxos require participants from a majority of acceptors (Property 2). Any two majorities of acceptors will intersect, in other words they will have at least one acceptor in common. □

We can build upon lemma 11 to show the following:

Lemma 12 (Weakened safety of future proposals)**.** *If a value v is decided in epoch e and value w is proposed in f where $f > e$ then w must have been proposed in g where $e \leq g < f$*

Proof of lemma 12. Assume value v is decided in epoch e and value w is proposed in f where $f > e$.

The proposer in f will have proposed w after completing phase one and choosing w as a result of the value selection rules.

From lemma 11, at least one acceptor must have sent both *accept(e,v)* to the proposer in e and *promise(f,_,_)* to the proposer in f since $e < f$.

From lemma 10, we know that this acceptor sent *accept(e,v)* prior to sending *promise(f,_,_)* as $e < f$.

Before the acceptor sent *accept(e,v)*, they will have set their last promised epoch (Property 7) and last accepted epoch to e and set the last accepted value to v (Property 9).

Since the last promised epoch was set to e and it is monotonically increasing (lemma 6), then the acceptor could only have accepted proposals for $\geq e$ after sending *accept(e,v)*. Conversely, before sending *promise(f,_,_)* the acceptor could only have accepted proposal for $\leq f$ (from lemma 8). Therefore, the proposer will only have updated its last accepted value for a proposal from e to f. Therefore the acceptor will have sent *promise(f,g,x)* where $e \leq g < f$ and x is the value proposed in g.

According to the value selection rules (Property 4), the proposer in f will only not choose proposal x if it also receives a proposal with a higher epoch which must also be $< f$ so either way w must have been proposed in g where $e \leq g < f$ □

Using lemma 9 and considering the case that $f = succ(e)$ in lemma 12 then $g = e$ so it follows that:

Corollary 12.1 (Base case for safety of future proposals)**.** *If the value v is decided in epoch e and the value w is proposed $succ(e)$ then $v = w$.*

Definition 5. *We say that an epoch e is limited to value v if e must decide on v if a decision is reached.*

Therefore, corollary 12.1 can also be stated as if v is decided in epoch e then $succ(e)$ is limited to v.

Corollary 12.1 can be extended as follows:

Corollary 12.2 (Inductive case for safety of future proposals)**.** *If the value v is decided in epoch e and the proposals from e (exclusive) to f (inclusive) are limited to the value v then if value w is proposed in g such that $g = succ(f)$ then $v = w$.*

Theorem 13 (Safety of future proposals)**.** *If the value v is decided in epoch e and the value w is proposed in epoch f such that $e < f$ then $v = w$*

Theorem 13 specifies that once a value is decided in epoch e then all subsequent epochs $> e$ which reach a decision, will decide upon the same value.

Proof of theorem 13. Assume the value v is decided in epoch e. We will prove this by induction.

Firstly, we will demonstrate that no proposer will propose a different value using proposal $succ(e)$. We cannot know if a decision will be reached in epoch $succ(e)$ but if one is reached, it will always decide on v, the same value as e. In other words, the successor of proposal e is limited to v.

(Base case) If the value w is proposed in epoch f such that $f = succ(e)$ then $v = w$.

This was proven by corollary 12.1.

Next, we will demonstrate that the successor of a sequence of limited proposals following a decided proposal is also limited to the same value.

(Inductive case) If the proposals from e to f are limited to the value v then if value w is proposed in epoch g such that $g = succ(f)$ then $v = w$.

This was proven by corollary 12.2.

By induction, we see that if the value v is decided in epoch e then all subsequent proposals will be limited to value v. Therefore proving theorem 13 and thus theorem 14.

□

Proof of safety of Classic Paxos

Overall, to prove the safety of Paxos, we show the following:

Theorem 14 (Safety for Classic Paxos)**.** *If the value v is decided in epoch e and the value w is decided in epoch f then $v = w$*

This could also be stated as if a value v is decided then all epochs are limited to v.

Proof of theorem 14. Consider the case that $e = f$.

Lemma 9 shows that at most one value will be proposed with any given epoch. Since it is necessary for a value to be proposed before it is decided, this means that at most one value can be decided with any epoch too.

Consider the case that $e \neq f$.

Since there is a total ordering on epochs then either $e < f$ or $e > f$. From the symmetry of theorem 14, we can assume $e < f$ and derive $e > f$ by swapping e and f.

For a value to be decided, it must first be proposed, therefore a stronger theorem is theorem 13. □

Now that we have proven the safety of the decided values, we will prove that only decided values will be returned by the proposers.

Lemma 15 (Safety of learning)**.** *If the value v is returned by a proposer then v has been decided.*

Proof of Lemma 15. Consider a proposer p who has returned v.

Prior to returning v, p must have received *accept(e)* for some epoch e from a majority of acceptors (Property 3).

We know *accept(e)* must have been sent in response to *propose(e,v)* from proposer p (Property 1).

Therefore the majority of acceptors must have accepted the proposal (e, v) so by definition the value v must have been decided (Property 9). □

Result	Properties:	Other results:
Monotonicity of promises (6)	6, 7, 10	
Relation between acceptor epochs (7)	8, 10	6
General promise format (8)	8	7
Value uniqueness (9)	1	
Message ordering (10)	6, 7, 8, 9	6
Quorum intersection (11)	2	
Weakened safety of future proposals (12)	4, 7, 9	11, 8, 10, 6
Base case for safety of future proposals (12.1)		9, 12
Inductive case for safety of future proposals (12.2)		9, 12
Safety of future proposals (13)		12.1, 12.2
Safety for Classic Paxos (14)		9, 13
Safety of learning (15)	1, 3, 9	

Table 2.3: Use of algorithm properties to prove the safety of Classic Paxos

Table 2.3 outlines how we divided up our proof of Safety for Classic Paxos (Theorem 14). Our approach of using multiple layers of intermediate results will allow us to revise this proof throughout this thesis, without reproducing the complete proof.

It is worthwhile noting that lemmas 6, 7, 8 and 10 are properties of the acceptor algorithm for Classic Paxos. Their proofs do not rely upon any properties of the proposer algorithm and thus these lemmas still hold if the proposers behave arbitrarily. Likewise lemmas 11 and 9 are properties of the proposer algorithm for Classic Paxos and do not reply upon the acceptor algorithm.

2.7 Progress

The proof of safety for Classic Paxos does not depend on any liveness conditions such as bounded message delay or execution time. In contrast, the proof of progress, the subject of this section, must depend upon some liveness conditions, as proved by the FLP result [FLP85]. We will formulate progress as follows: from time 0 to Global Stabilisation Time (GST), a system of participants have been executing Classic Paxos. No assumptions regarding liveness are made during this time. The system may be in any reachable state at GST.

From GST, the following *liveness conditions* must apply for a sufficient period:

- At least a majority of acceptors are live and reply to messages from proposers, if specified by the algorithm, within the known upper bound δ_a[16].
- Exactly one (fixed) proposer is live and its relative clock is no faster than δ_d ahead of global time. We assume that no messages from other proposers are delivered[17].
- Messages between the proposer and majority of acceptors are delivered within the known bound δ_m.

This model of initial asynchrony, eventually followed by synchrony is sometimes known as partial synchrony [DLS88].

As expected, we require that a majority of acceptors are up and able to communicate as the proposer will need to get majority agreement to complete the two phases of Classic Paxos. We also need to require that exactly one proposer is executing the algorithm to prevent proposers duelling indefinitely, as illustrated in Figure 2.6 (§2.3). The requirements for bounded execution time, message delay and clock drift are to ensure that the acceptors will have a chance to respond to messages from the proposer prior to the proposer restarting the proposal.

[16]This need not be a fixed group, but we will assume it is to simplify our proof.

[17]This assumption is not necessary for guaranteeing progress but it does simplify our proof.

Theorem 16. *Provided the liveness conditions are satisfied, a proposer will eventually terminate and return a value v.*

Proof of Lemma 16. Consider a system which has reached GST. The proposer p may be at any stage of the proposer algorithm.

Consider the case that p is at the start of the proposer algorithm.

The proposer p will generate an epoch $e \in E$ and dispatch *prepare(e)* to all acceptors. It follows from the liveness conditions that the majority of acceptors will receive *prepare(e)* within δ_m. Likewise, if e is greater than or equal to an acceptor's last promised proposal number then it will promise within δ_a. Otherwise, the acceptor will not reply to the prepare message. Any promises sent by acceptors will be received within δ_m.

If the proposer does not receive promises from a majority of acceptors after $\delta_a + 2\delta_m + \delta_d$, the proposer will abandon epoch e and restart the proposer algorithm. The proposer p will generate a new epoch f where $f > e$ (Property 5) and repeat phase one. As the acceptors will not receive any messages from other proposers, the last promised proposal number will not increase, except in response to p.

Eventually, the proposer p will have generated a sufficiently large epoch that the majority of the acceptors will promise within $\delta_a + 2\delta_m$. The proposer will proceed to choosing a value.

If no accepted proposals were returned by acceptors then the proposer is free to choose its own value. Otherwise, the proposer must choose the value associated with the highest epoch. Since the epochs are totally ordered and values are unique to epochs (Lemma 9.1) then proposer will always be able to choose a value v.

The proposer then dispatches *propose(e,v)* to the acceptors and it is received within δ_m. Since the acceptors will not have updated their last promised epoch (as there are no other proposers) then the acceptors will accept the proposal. Since the proposer will receive accepts from the majority of acceptors within $\delta_a + 2\delta_m$, the value v is returned.

Consider that case that p is elsewhere in the proposer algorithm.

If the proposer p is in phase one of the proposer algorithm then it will proceed as described in the first case. If the proposer is in phase two of the proposer algorithm then it may timeout if its epoch is less then the majority of acceptors. In this case, the proposer will not receive accepts from the majority of acceptors before $\delta_a + 2\delta_m + \delta_d$, thus it will abandon the proposal and restart the proposer algorithm, as described in the first case. □

From Lemma 15, a weaker form of Lemma 16 is:

Corollary 16.1. *Provided the liveness conditions are satisfied, a value v will eventually be decided.*

Note how these liveness conditions differ from the liveness conditions for SAA (§5). SAA requires that the single acceptor is up whereas Classic Paxos requires a majority of acceptors to be up. Classic Paxos, however, requires exactly one proposer is up, whereas SAA only requires that at least one proposer is up. Classic Paxos also requires bounds on execution time (for acceptors only) and message delay, unlike SAA which only requires eventual execution and message delivery.

This proof of progress requires that proposers know the bounds δ_a, δ_m and δ_d. If the proposer does not wait long enough before retrying then the system may not make progress. If the bound is unknown, this can be addressed using backoff timers as new epochs are generated.

2.8 Summary

Single-valued distributed consensus is the problem of deciding a single value between a set of participants. An algorithm is said to solve distributed consensus provided it guarantees safety, so that decisions are final and progress, so that eventually a decision will be reached. Algorithms operating in an asynchronous, unreliable distributed system cannot guarantee progress without assumptions regarding the liveness and/or synchrony of the system.

	SAA	**Classic Paxos**
Number of acceptors	1	n_a
Conditions for progress:		
Number of live proposers	1 or more	exactly 1
Number of live acceptors	All (1)	$\lfloor n_a/2 \rfloor + 1$ or more
Synchrony	no	yes
Number of messages	2	$2n_a + 2$ or more
Number of round trips	1	2 or more
Number of persistent writes	1	3 or more

Table 2.4: Comparison between SAA & Classic Paxos

This chapter introduced two known distributed algorithms: the Single acceptor algorithm (SAA) and Classic Paxos. Both algorithms guarantee safety and progress thus both solve distributed consensus, however, their liveness conditions for progress differ. Both algorithms divide participants in a system into proposers, which propose values to be decided and acceptors, which choose and store values. SAA requires that the single acceptor and at least one proposer are live. Classic Paxos requires that a strict majority of acceptors and exactly one proposer is live and that these participants are operating synchronously. Under these conditions, a proposer in SAA is guaranteed to terminate in one round trip to the acceptor, whereas, a proposer in Classic Paxos is guaranteed to terminate in a finite

number of steps, with a minimum of two round trips to a strict majority of acceptors. These differences are summarised in Table 2.4.

Classic Paxos has been the subject of extensive study in recent decades and the next chapter discusses the wide range of consensus algorithms within the Paxos family.

Chapter 3

Known revisions

Thus far, we have considered Classic Paxos as a single concrete algorithm to solve single valued distributed consensus. Instead however, Paxos is a broad family of algorithms for distributed consensus. In this systematisation of knowledge chapter, we survey some of the most commonly used refinements to the Classic Paxos algorithm.

3.1 Negative responses (NACKs)

Classic Paxos as has been detailed so far, could be described as following the idea that "If you can't say something nice, don't say nothing at all"[1]. More specifically, acceptors will not reply to proposers whose epoch e is less than their last promised epoch e_{pro}. The result is that proposers must wait for their prepare to timeout and retry with a new epoch.

This can be improved by adding negative responses, such as *no-promise(e)* and *no-accept(e)*. These negative responses would be sent by acceptors to proposers upon receipt of prepare or propose messages where $e < e_{pro}$. When a proposer receives negative responses, it can opt to restart the proposal with a higher epoch. Otherwise, the proposer can ignore the negative responses and wait to see if they receive positive responses from a majority of participants. If a proposer receives negative responses from a majority of acceptors, then its proposal will not be successful and the proposer should restart the proposal. It is safe for a proposer to abandon or restart a proposal at any stage, since this is functionally equivalent to a proposer failing and restarting.

The acceptors can include additional information in the negative responses such as the *no-promise(e,f)* and *no-accept(e,f)*, where f is the acceptor's last promised epoch or even *no-promise(e,f,g,v)* and *no-accept(e,f,g,v)*, where (g, v) is the acceptor's last accepted proposals[2].

[1] Quote from Thumper in the Disney film Bambi.

[2] For example, in the Raft algorithm acceptors include their last promised epoch (referred to as *cur-*

Algorithm 5: Proposer algorithm for Classic Paxos with NACKs

```
1  v, e_max ← nil
2  Q_P, Q_A ← ∅
3  e ← min(ℰ)
4  ℰ ← ℰ \ {e}
   /* Start Phase 1 for epoch e */
5  send prepare(e) to acceptors
6  while |Q_P| < ⌊n_a/2⌋ + 1 do
7      switch do
8          case promise(e,f,w) received from acceptor a
9              Q_P ← Q_P ∪ {a}
10             if f ≠ nil ∧ (e_max = nil ∨ f > e_max) then
11                 e_max ← f, v ← w
12         case no-promise(e,f) received from acceptor
               /* abandon e and restart with epoch > f */
13             ℰ ← {n ∈ ℰ|n > f}
14             goto line 1
15 if v = nil then
16     v ← γ
   /* Start Phase 2 for proposal (e,v) */
17 send propose(e,v) to acceptors
18 while |Q_A| < ⌊n_a/2⌋ + 1 do
19     switch do
20         case accept(e) received from acceptor a
21             Q_A ← Q_A ∪ {a}
22         case no-accept(e,f) received from acceptor
               /* abandon e and restart with epoch > f */
23             ℰ ← {n ∈ ℰ|n > f}
24             goto line 1
25 return v
```

Algorithms 5 and 6 give an example of how this might work in practice. The lines in grey are unchanged from the Classic Paxos proposer and acceptor algorithms. If the proposer receives either *no-promise(e,f)* or *no-accept(e,f)* then it restarts the algorithm and skips over all epochs $\leq f$ as these are unlikely to be successful (Algorithm 5, lines 12-14 & 22-24).

Figure 3.1 gives an example of Algorithms 5 & 6 in practice. In this scenario, initially

rent term) in negative responses to prepare and propose messages (referred to as *AppendEntries* and *RequestVote* respectively) [OO14, Figure 2].

Algorithm 6: Acceptor algorithm for Classic Paxos with NACKs

```
1  while true do
2      switch do
3          case prepare(e) received from proposer
4              if e_pro = nil ∨ e ≥ e_pro then
5                  e_pro ← e
6                  send promise(e,e_acc,v_acc) to proposer
7              else
                   /* e < e_pro so reply with NACK */
8                  send no-promise(e,e_pro) to proposer
9          case propose(e,v) received from proposer
10             if e_pro = nil ∨ e ≥ e_pro then
11                 e_pro ← e
12                 v_acc ← v, e_acc ← e
13                 send accept(e) to proposer
14             else
                   /* e < e_pro so reply with NACK */
15                 send no-accept(e,e_pro) to proposer
```

proposal $(5, B)$ has been accepted by all three acceptors (and thus has been decided). The proposer p_1 begins phase one by dispatching *prepare(0)* to all acceptors. In Classic Paxos, this proposer would need to wait for a timeout and retry with proposal numbers 2, 4 and 6, thus requiring at least 4 round trip times. However, with NACKs the acceptor a_1 can notify the proposer that its last proposed proposal number is 5 and the proposer p_1 thus skips proposal numbers 2 and 4, allowing phase one to be completed in 2 round trips.

NACK's have replaced timeouts as we assume that messages are eventually delivered. We can therefore remove the synchrony assumptions from our progress proof. However, we still require there to be exactly one proposer for guaranteed progress, which we will later show can be implemented assuming synchrony (§3.4).

Note that these two optimisations of restarting a proposal and skipping over epochs are distinct and could be used separately. For example, a proposer recovering after a long failure could opt to skip over some epochs to increase its likelihood of completing the proposer algorithm during its first try. It is also worthwhile noting that NACKs need not include the proposer's epoch nor do we need separate messages for each phase. In fact, our existing accept message could be used for this purpose instead. We have chosen this approach for consistency with the existing messages and to ensure each message serves a single clearly defined purpose.

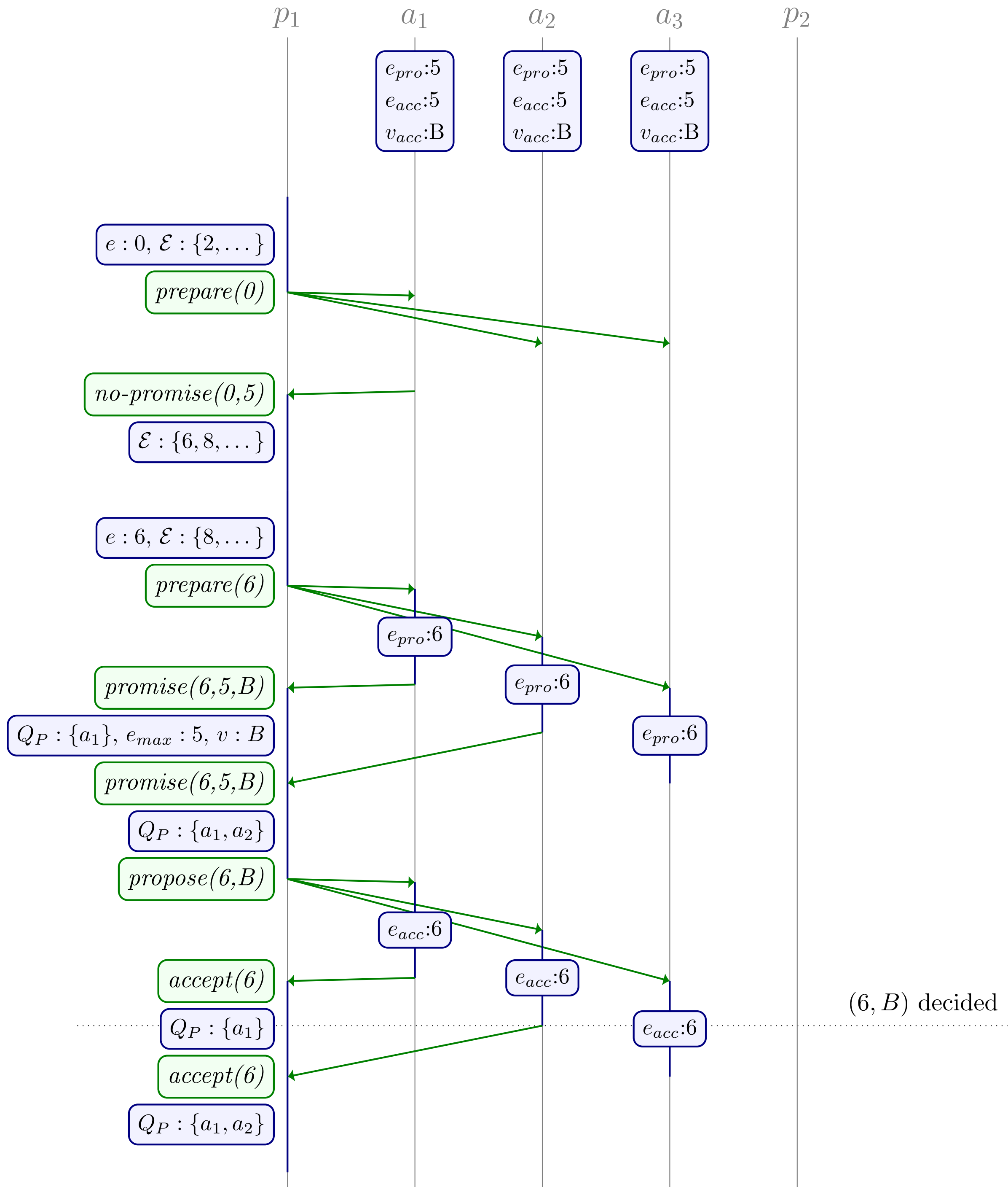

Figure 3.1: Classic Paxos with NACKs (Alg. 5,6)

3.2 Bypassing phase two

The proposer algorithm for Classic Paxos is doing more work then is strictly necessary to meet the requirements of distributed consensus. In practice, if a proposer learns that a value has already been decided, because a majority of acceptors return the same proposal during phase one, then the proposer may skip phase two and return the value in the proposal.

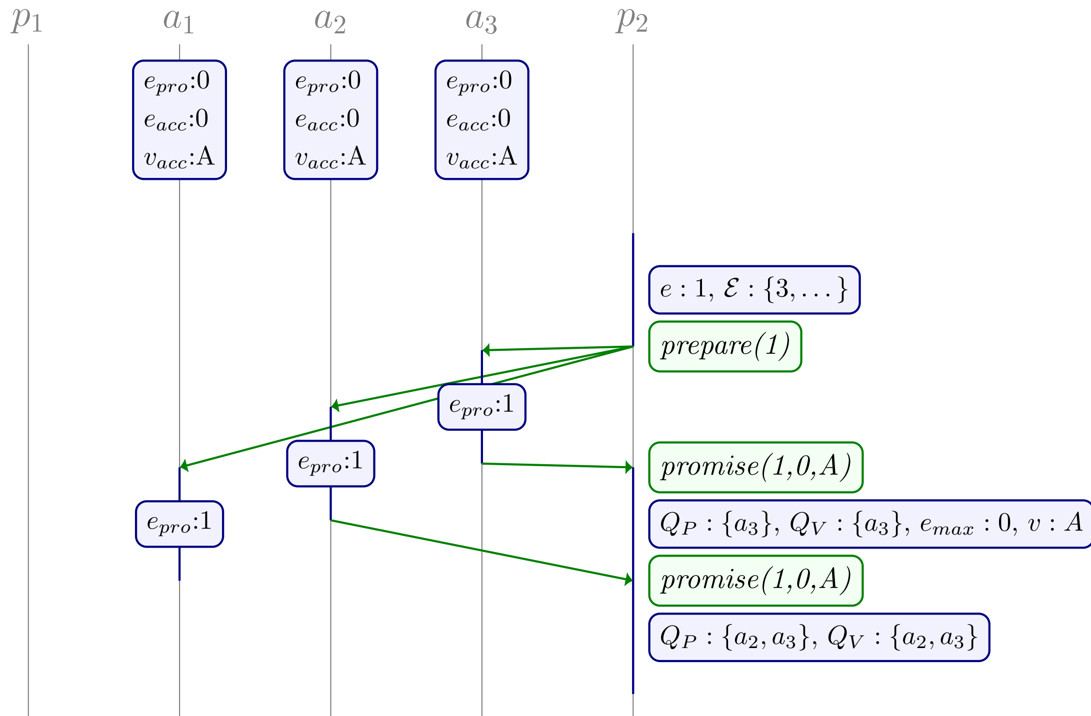

Figure 3.2: Classic Paxos with bypass (Alg. 4,7)

There are therefore three possible outcomes of the proposer's phase one:

Decision not reached - No proposals were received with promises during phase one, therefore no value has yet been decided. The proposer will propose its candidate value in phase two.

Decision reached - All promises received in phase one agreed on a value. This value has been decided and the proposer has learned the chosen value. No further action is necessary.

Uncertainty - Some proposals were returned in phase one. The proposer is uncertain if commit point has been reached. If reached, then the decided value is the value returned with the highest epoch so the proposer therefore proposes this value.

Algorithm 7 gives a version of the proposer algorithm which bypasses phase two when it learns that a decision has been reached. This is achieved by maintaining a set of acceptors, Q_V, who have promised and returned the proposal (e_{max}, v) with their promise (lines 3,12,15,16). Once phase one is completed, if Q_V includes the majority of acceptors then phase two can be bypassed (lines 18-19). Note that the acceptor algorithm is unchanged.

Figure 3.2 demonstrates how our first Classic Paxos example (Figure 2.2) could be improved using phase two bypass. Proposer p_2 is able to skip over phase two as it learns that the proposal $(0, A)$ has already been decided since it has been accepted by a majority of acceptors.

Algorithm 7: Proposer algorithm for Classic Paxos with phase two bypass

```
state:
  • Q_V: set of acceptors who have promised with (e_max, v)

1  v, e_max ← nil
2  Q_P, Q_A ← ∅
3  Q_V ← ∅
4  e ← min(ℰ)
5  ℰ ← ℰ \ {e}
   /* Start Phase 1 for epoch e */
6  send prepare(e) to acceptors
7  while |Q_P| < ⌊n_a/2⌋ + 1 do
8      switch do
9          case promise(e,f,w) received from acceptor a
10             Q_P ← Q_P ∪ {a}
11             if f ≠ nil then
12                 if e_max = nil ∨ f > e_max then
13                     Q_V ← {a}
14                     e_max ← f, v ← w
15                 else if f = e_max then
16                     Q_V ← Q_V ∪ {a}
17         case timeout
18             goto line 1
19 if |Q_V| ≥ ⌊n_a/2⌋ + 1 then
       /* proposer has learned that (e_max, v) is decided */
20     return v
21 else
22     if v = nil then
23         v ← γ
       /* Start Phase 2 for proposal (e,v) */
24     send propose(e,v) to acceptors
25     while |Q_A| < ⌊n_a/2⌋ + 1 do
26         switch do
27             case accept(e) received from acceptor a
28                 Q_A ← Q_A ∪ {a}
29             case timeout
30                 goto line 1
31     return v
```

We can increase the likelihood of phase two bypass using the following techniques:

- If a proposer has received many of the same proposals in phase one but not quite reached the $\lfloor n_a/2 \rfloor + 1$ copies needed to bypass phase two then it may opt to wait for further promises before proceeding. The timeout would be required to limit this wait to maintain progress guarantees.[3]
- A proposer can concurrently start phase two and continue waiting for promises in phase one. If sufficient promises with the same proposal are received before phase two is completed then the remainder of phase two can be bypassed.
- Instead of tracking whether the greatest proposal is returned by the majority, proposers could track all proposals returned.
- Proposers could re-use promises from previous epochs when tracking the proposals returned for phase two bypass. This could involve storing previously received proposals to persistent storage, however this is not necessary.
- Proposers could include proposals from NACKs, again regardless of epoch or message name, when tracking the proposals returned for phase two bypass.
- Acceptors could store all accepted proposals instead of just the last accepted proposal. Acceptors could then include all previously accepted proposals in promise messages (and NACKs), providing proposers with additional information about the state of the system[4].

3.3 Termination

In Classic Paxos, even with bypassing phase two enabled, a proposer must communicate with a majority of acceptors to learn the decided value. This means that the liveness conditions for progress are necessary as well as sufficient for progress. In other words, regardless of the state of the system, the majority of acceptors must be up and communicating for a proposer to execute its algorithm and return a decided value.

We can improve this by adding an optional phase three to Classic Paxos in which the acceptors learn the value has been decided. The acceptors can then notify future proposers that the value has been decided, enabling the proposer to return a decided value without waiting upon the majority of acceptors. Adding phase three to Classic Paxos serves an important purpose that may not be immediately apparent, namely that the liveness conditions are no longer necessary for progress. With this variant, Classic Paxos can make progress provided either a majority of acceptors are up or at least one acceptor who has

[3]In the previous section on negative responses (§3.1), we saw that a proposer may opt to retry a proposal early, prior to timing out. We now see that a proposer may opt to wait longer before proceeding to phase two.

[4]Some papers such as [VRA15] described this approach as Paxos and describe storing only the last accepted proposals as an optimisation.

Algorithm 8: Proposer algorithm for Classic Paxos with termination

```
1  v, e_max ← nil
2  Q_P, Q_A ← ∅
3  e ← min(ℰ)
4  ℰ ← ℰ \ {e}
   /* Start Phase 1 for epoch e */
5  send prepare(e) to acceptors
6  while |Q_P| < ⌊n_a/2⌋ + 1 do
7      switch do
8          case promise(e,f,w) received from acceptor a
9              Q_P ← Q_P ∪ {a}
10             if f ≠ nil ∧ (e_max = nil ∨ f > e_max) then
11                 e_max ← f, v ← w
12         case decided(w) received from acceptor
               /* skip the remainder of phase one & phase two */
13             v ← w
14             goto line 29
15         case timeout
16             goto line 1
17 if v = nil then
18     v ← γ
   /* Start Phase 2 for proposal (e,v) */
19 send propose(e,v) to acceptors
20 while |Q_A| < ⌊n_a/2⌋ + 1 do
21     switch do
22         case accept(e) received from acceptor a
23             Q_A ← Q_A ∪ {a}
24         case decided(w) received from acceptor
               /* skip the remainder of phase two */
25             v ← w
26             goto line 29
27         case timeout
28             goto line 1
29 return v
30 /* Start of Phase 3 for decided value v */
31 send decided(v) to acceptors
```

Algorithm 9: Acceptor algorithm for Classic Paxos with termination

state:

- v_{dec}: decided value

```
1  while true do
2      switch do
3          case prepare(e) received from proposer
4              if v_dec ≠ nil then
                   /* notify proposer that decision has been reached */
5                  send decided(v_dec) to proposer
6              else if e_pro = nil ∨ e ≥ e_pro then
7                  e_pro ← e
8                  send promise(e,e_acc,v_acc) to proposer
9          case propose(e,v) received from proposer
10             if v_dec ≠ nil then
                   /* notify proposer that decision has been reached */
11                 send decided(v_dec) to proposer
12             else if e_pro = nil ∨ e ≥ e_pro then
13                 e_pro ← e
14                 v_acc ← v, e_acc ← e
15                 send accept(e) to proposer
16         case decided(v) received from proposer
               /* save decided value */
17             v_dec ← v
```

been notified of decision is up. As a result, the proposer may return a decided value after communicating with just one acceptor. Algorithms 8 and 9 provide an example of how this could be implemented into Classic Paxos.

Algorithm 9 adds v_{dec}, the *decided value* state to acceptors. In algorithm 8, once a proposer learns that a value v is decided, it sends *decided(v)* to all acceptors[5]. Upon receipt of *decided(v)*, an acceptor can set the decided value v_{dec} to v and henceforth reply to incoming messages (regardless of message type or epoch) with *decided(v)*. All other state on the acceptor can now be safely discarded. This approach is taken by algorithms such as Mencius [MJM08, §4.2].

Figure 3.3 demonstrates how this additional phase can allow future proposers (in this case p_2) to learn the decided value after communicating with just one acceptor a_3. Figure 3.3 uses the same scenario as our first Classic Paxos example (Figure 2.2).

This approach requires the proposer to send the value, which could be large, to all acceptors,

[5]This message is sometimes called *learn*.

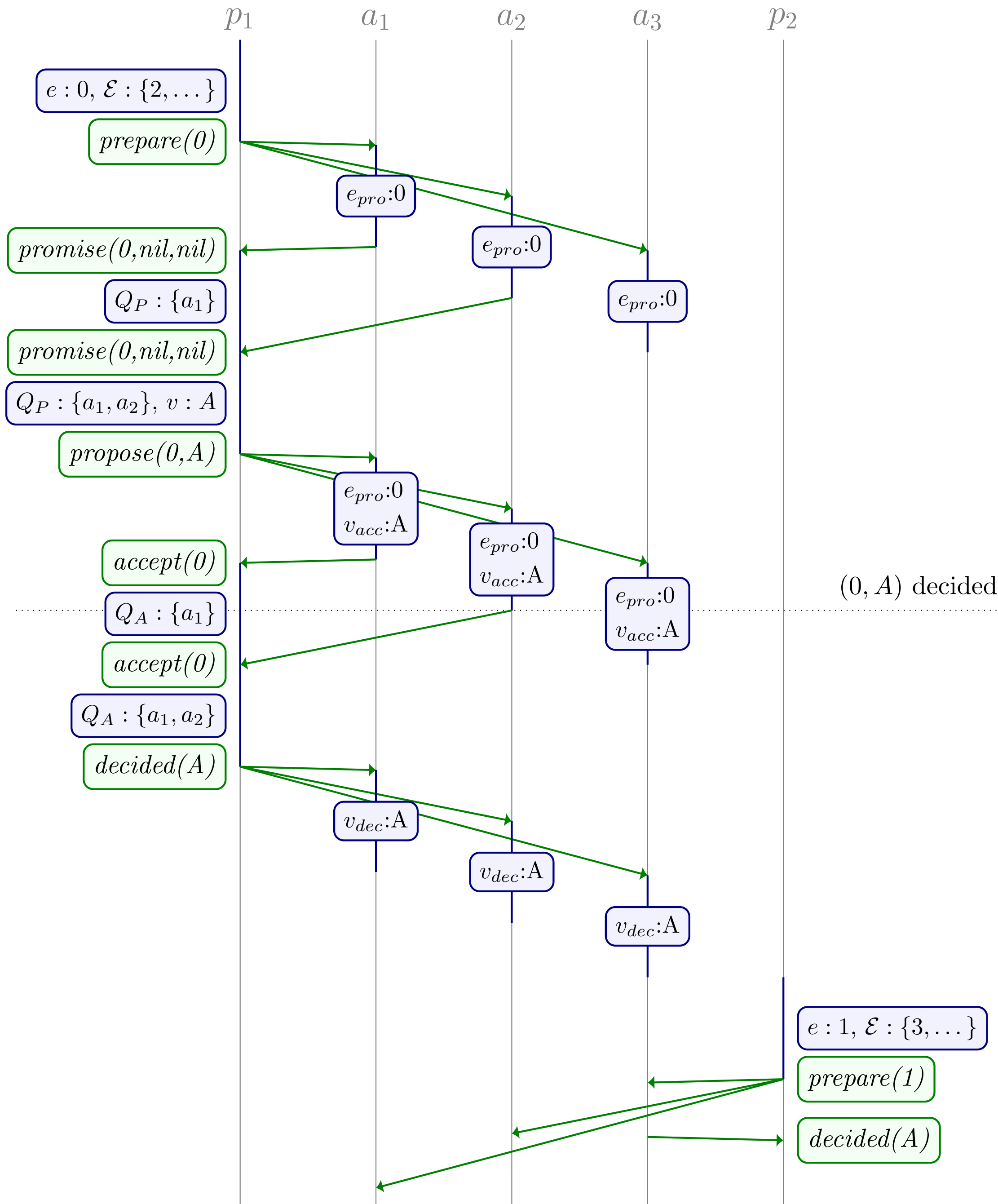

Figure 3.3: Classic Paxos with termination (Alg. 8,9)

despite the fact that at least a majority of the acceptors will already have a copy of value. Alternatively, the proposer could send *decided(e)* where e is the epoch in which the value v was decided. If an acceptor receives *decided(e)* and has accepted a proposal from e (or a subsequent epoch) then the acceptor learns that the value in that proposal has been decided. This follows from Value Uniqueness (Lemma 9) and the Safety of future proposals (Theorem 13).

Alternatively, if a proposer learns that a decision has been reached but does not know the decided value then it can learn the decided value by executing phase one of Classic

Paxos. From the Safety of future proposals (Theorem 13), if a decision has been reached, the value chosen by the value selection rules will be the decided value.

Most consensus algorithms utilise some form of termination. Mencius [MJM08] and Ring Paxos [MPSP10, §4] use explicit phase three messages (called *learn* and *decision* respectively). Whereas algorithms such as Raft and VRR [LC12, §4.1] use a commit index in future messages instead of an explicit phase three to notify acceptors that a decision has been reached. Once an acceptor learns that a proposal has been decided, it is safe for this information to be shared with other participants including acceptors [MJM08, §4.5].

3.4 Distinguished proposer

In Figure 2.6 (§2.3), we observed the issue of duelling proposers where multiple proposers conflict over the proposal to be decided. This problem of duelling proposers is why our proof of progress (§2.7) assumed that exactly one proposer was executing Classic Paxos.

In practice, algorithms can minimise the likelihood of duelling proposers by designating one proposer as the *distinguished*. By modifying the proposer algorithm of non-distinguished proposer to forward candidate values to this proposer, the distinguished proposer becomes a point of serialisation, minimising the chance of duelling[6]. This mechanism improves reliability of performance by making it more likely that exactly one proposer is executing Classic Paxos at a given time.

If the distinguished proposer appears to be slow or unresponsive, another proposer can become a distinguished proposer and thus propose values directly. It is always safe for there be to no distinguished proposer, multiple distinguished proposers or inconsistent knowledge of the distinguished proposer[7]. However to guarantee progress, there should be exactly one distinguished proposer at a given time and all proposers should be aware of its identity. To satisfy this condition, we require reliable failure detection, which is not possible in an asynchronous distributed system [FLP85]. Instead we can approximate reliable failure detectors with heartbeats and timeouts, this does however require us to strengthen the liveness conditions for progress to bound message delay, clock drift and operating speed between proposers. The weakest liveness conditions for failure detection are studied elsewhere in the literature [CHT96, MOZ05].

This optimisation, known as the distinguished proposer, is widely utilised [Lam01a, §2.4][LC12,

[6]In practice, candidate values often originate from external clients who can try to send values directly to the distinguished proposer. Examples of algorithms which take this approach include VRR [LC12, §4] and Raft [OO14, §8]. Alternatively, clients can broadcast values to all proposers, an approach taken by Moderately Complex Paxos [VRA15, §2.1]

[7]In other words, choosing a distinguished proposer is not a leader election problem and does not itself require distributed consensus

§4.2][OO14, §5.1][VRA15, §3][MPSP10, §3], usually in combination with Multi-Paxos (discussed in §3.6).

3.5 Phase ordering

During phase one of Classic Paxos, the proposer does not require knowledge of the value γ which they will propose if possible, in phase two. It is therefore possible for a proposer to execute phase one prior to knowledge of a value to propose. When the proposer then learns the value to propose, it may now decide this value in one round trip instead of two, provided no other proposer has also executed the proposer algorithm for a greater epoch. We can increase the likelihood that this will occur by also making this proposer a distinguished proposer.

This observation is widely utilised [Lam01a, §3][MPSP10, §4], usually in combination with a distinguished proposer and Multi-Paxos (discussed in §3.6)

3.6 Multi-Paxos

Thus far, we have considered how to reach consensus over a single value. In practice, these algorithms are usually utilised to reach consensus over an infinite sequence of values. Broadly speaking, we can divide existing algorithms for consensus over a sequence into two families:

Classic Paxos algorithms which are based upon executing multiple distinct instances of single valued consensus. Examples include Classic Paxos, Mencius [MJM08] and Fast Paxos [Lam05a]. These approaches are rarely used in production systems.

Multi-Paxos algorithms where one proposer takes the role of *leader* by executing phase one over the sequence and then coordinates decisions until a new leader takes over. This approach is widely utilised in production systems. Examples include Chubby [Bur06, CGR07], Zookeeper [HKJR10, JRS11], Ring Paxos [MPSP10], Viewstamped Replication [OL88, LC12] and Raft [OO14].

Multi-Paxos is an optimisation of Classic Paxos for consensus over a sequence. Multi-Paxos differs from successive instances of Classic Paxos in one key way. The phase one of Classic Paxos is shared by all instances. Each acceptor needs only to store the last promised epoch once. Prepare and promise messages are not instance-specific and therefore do not need an index included in the phase one messages.

This is combined with distinguished proposer (§3.4) and phase ordering (§3.5) optimisations as follows. Phase one is executed by a proposer prior to knowledge of the values to propose.

Once phase one is completed, we will refer to this proposer as the *Leader*[8]. The leader is the distinguished proposer and thus is responsible for reaching decisions. If another proposer suspects that the leader has failed, the proposer can take over as leader by executing phase one, we refer to this process as *leader election*[9] and thus become the next distinguished proposer.

The key advantages of Multi-Paxos is that during steady state, each decision is reached in one round trip to the majority of acceptors and one synchronous write to persistent storage. The system is in steady state when exactly one of the proposers (the leader) is in the replication phase and a majority of acceptors are up and responsive. A system should be operating in this state most of the time.

Multi-Paxos places substantial load on the leader. In the steady state, this single proposer is responsible for receiving candidate values, assigning values to indexes, proposing values to acceptors, collecting accept messages, learning decided values and notifying participants of decisions. For this reason, the leader is often the bottleneck in Multi-Paxos systems. This unbalanced approach leads to high pressure on the leader and its network links, whilst leaving the other participants and other areas of the network under-utilised. Furthermore, whilst the system is now able to achieve consensus in one round trip instead of two, there is also only one proposer who can achieve this. Candidate values must therefore be forwarded to the leader (or clients redirected) which can add an additional round trip. These reasons are the motivation for algorithms such as Mencius [MJM08, §3].

3.7 Roles

So far in this thesis, we have divided the responsibilities in Classic Paxos into two distinct roles: acceptors and proposer[10]. This approach was adopted as it is widely used in the academic literature, however this distinction is also quite arbitrary.

We could for example have only one role called *replica* which co-locates proposer and acceptor into one participant. The replica would benefit from a reduction in the number of acceptors it needs to communicate with by one and the proposer could use the acceptors last promised epoch when generating the next epoch.

This approach is widely discussed in the academic literature and adopted in practice, examples include Simple Paxos [Lam01a, §3], Chubby [CGR07], Mencius [MJM08], VRR [LC12], Raft [OO14] and Moderately Complex Paxos [VRA15, §4.4].

[8]The leader is also known in the literature as master, primary [LC12] or coordinator [MPSP10]. Non-leader proposers are also known as backups [LC12] and followers [OO14, §5.1]. The term leader here should not be confused with leaders, which is sometimes used as an alternative term for proposers, for example by Renesse and Altinbuken [VRA15].

[9]This is referred to *view change* in Viewstamped Replication [OL88, LC12]

[10]Proposer can be subdivided into distinguished and non-distinguished or leaders and non-leaders

Conversely, we could increase the number of roles, for example we could add a *reader* role, a participant who asks acceptors for the last accepted proposal to try to determine if the value has been decided and what that value is or a *recovery proposer* who acts like a proposer except if no values are returned in phase one then they exit instead of returning a value.

3.8 Epochs

Earlier we specified that epochs, E, are required to be an infinite totally ordered set (Definition 2) and that each proposer should be given a disjoint subset of the epochs. Our pseudocode remains general and does not specific how epoch such be generated. Many different mechanisms could be utilised to allocate epochs.

The approach used in our examples is that epochs are natural numbers $E = \mathbb{N}^0$, which have been divided round robin between the proposers.

Alternatively, epochs could be lexicographically ordered tuples of the form (sid, pid), where sid is the proposal sequence number (persistent state) and pid is the unique proposer id (config). The current sid must be written to persistent storage before use by the proposer to ensure proposal uniqueness. Since sid is monotonically increasing, only the most recent sid need be stored[11].

Both these approaches require each proposal to begin with a synchronous write to persistent storage. This can be avoided by instead using epochs of the form (sid, pid, vid), where sid is the sequence number (volatile state), pid is the unique proposer id (config) and vid is the proposer version number (persistent state). Proposers must increment vid each time they restart to ensure uniqueness of epochs, without writing sid updates to persistent storage.

Alternatively, we can avoid most synchronous writes by writing an upper bound on epoch e to persistent storage and only updating it as needed[12]. Epochs could otherwise be of the form (sid, pid), where sid is stored in volatile state but an upper bound is stored on persistent state.

This approach of writing an upper bound on a epoch to persistent storage can also be applied to the last promised epoch on acceptors. This removes the need for a synchronous writing to persistent each time an acceptor promises.

In practice, the write to persistent storage does not need to be completed until the start of phase two to maintain proposal uniqueness. As such, updating $\mathcal{E}$ and execution of phase one can be completed concurrently, mitigating the latency of a synchronous write to persistent storage.

[11]Note that with this scheme, pid and sid would replace $\mathcal{E}$.

[12]This is equivalent to batch pre-executing the writes to persistent storage.

Various mechanisms for allocating epochs are demonstrated by algorithms such as Simple Paxos [Lam01a, §2.5], Chubby [CGR07], VRR [LC12, §4] and Moderately Complex Paxos [VRA15]

3.9 Phase one voting for epochs

It has long been known that Classic Paxos does not require that epochs are unique if acceptors require that a proposer's epochs be strictly greater than the last promised proposal. This means that at most one proposer will reach phase two for a given epoch, since reaching phase two requires a proposer to have already reached majority agreement for phase one, thus guaranteeing uniqueness.

Algorithm 10: Acceptor algorithm for Classic Paxos with voting

state:

- p_{lst}: last proposer promised to, $p_{lst} \in P$ (persistent)

```
1  while true do
2    switch do
3      case prepare(e) received from proposer p
4        if (e_pro = nil) ∨ (e > e_pro) ∨ (e = e_pro ∧ p = p_lst) then
5          e_pro ← e, p_lst ← p
6          send promise(e,e_acc,v_acc) to proposer
7      case propose(e,v) received from proposer
8        if e_pro = nil ∨ e ≥ e_pro then
9          e_pro ← e
10         v_acc ← v, e_acc ← e
11         send accept(e) to proposer
```

We can implement exclusive epochs by voting by adding the requirement to promises that if the epoch from the prepare message e is equal to the last promised epoch e_{pro} then the proposer p must be the same as the proposer who was previously promised p_{lst}. This revised acceptor algorithm is show in Algorithm 10. The proposer algorithm remains almost unchanged, except that proposers no longer need to be allocated a disjoint subset of epochs, thus proposer can use any epoch $\mathcal{E} = E$.

Figure 3.4 gives an example of Classic Paxos with voting. In contrast to our first Classic Paxos example (Figure 2.2), proposer p_2 initially uses proposal number 0 (instead of 1). Proposer p_2 times out as p_1 has already completed phase one for proposal number 0. The proposer p_2 then tries proposal number 1 and proceeds as before.

Recall that in the proof of safety of Classic Paxos we used the following lemma:

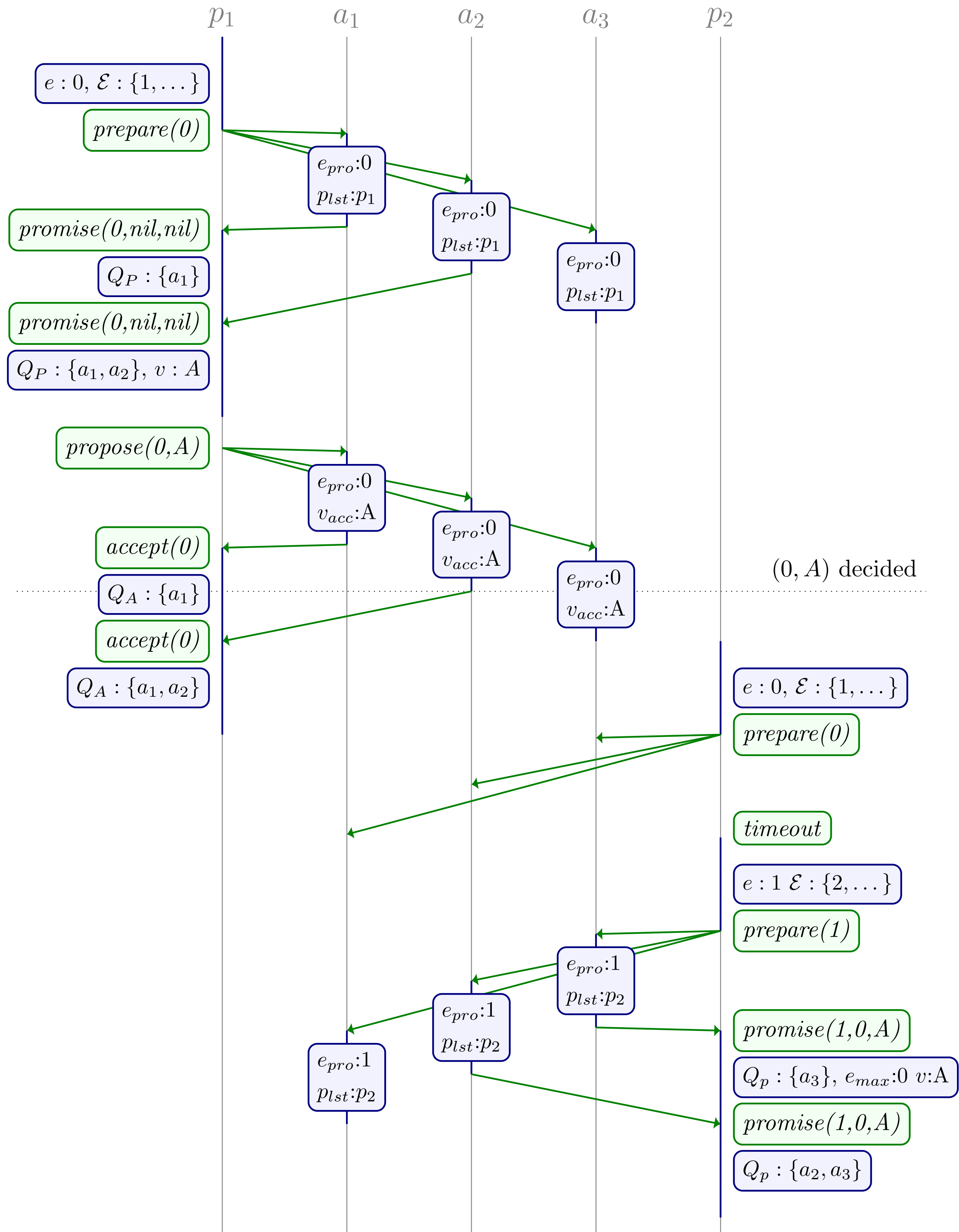

Figure 3.4: Classic Paxos with voting (Alg. 3,10)

Lemma 9 (Value uniqueness)**.** *If the value v is proposed in epoch e then no other value can also be proposed in e.*

We will now revise the proof of lemma 9 as follows.

Lemma 17. *Each acceptor will promise to at most one proposer for each epoch e.*

Proof of Lemma 17. Assume an acceptor has received *prepare(e)* and replied with *promise(e,...)*. The acceptor will have set its last promised epoch to e prior to sending the promise. Since the last promised epoch is monotonically increasing (lemma 6), then the acceptor's last promised epoch will henceforth be $\geq e$.

Assume the acceptor receives *prepare(e)* from another proposer. For the acceptor to promise in e it must be have case that $e >$ last promised epoch, yet last promised epoch is $\geq e$ thus the acceptor cannot accept the promise. □

Revised proof of lemma 9 using exclusive epochs by voting. It follows from Lemma 17 and the phase one quorum intersection requirement that:

Corollary 17.1. *At most one proposer will propose a value for a epoch.*

From this it follows that since each proposer will propose only one value for a given epoch then at most one value will be proposed for each epoch. □

Voting is used to allocate epochs in consensus algorithms such as Raft [OO14, §5.1].

3.10 Proposal copying

The pre-allocation of epochs in Classic Paxos (or exclusive access to epochs by voting) ensures that a unique proposer uses each epoch. It is important for safety (Lemma 9) to ensure that at most one value is used with each epoch. However, there is no requirement that only one proposer uses each epoch. Upon receipt of *propose(e,v)*, an acceptor learns two important facts. Firstly, that a proposer has successfully executed phase one with epoch e and secondly, that the outcome of the value selection rules was that value v was chosen to be associated with epoch e. Given this information, another proposer can not only reuse the proposal mapping (e, v) but they can also skip phase one and proceed directly to phase two by dispatching *propose(e,v)* to the acceptors. We refer to this technique as *proposal copying*[13]. Below are two examples of how a proposer may learn (and therefore copy) past proposals.

Algorithm 11: Proposer algorithm for Classic Paxos with proposal copying

1. $v, e_{max} \leftarrow nil$
2. $Q_P, Q_A \leftarrow \emptyset$
3. $e \leftarrow min(\mathcal{E})$
4. $\mathcal{E} \leftarrow \mathcal{E} \setminus \{e\}$
5. **send** *prepare(e) to acceptors*
6. **while** $|Q_P| < \lfloor n_a/2 \rfloor + 1$ **do**
7.   **switch do**
8.   **case** *promise(e,f,w) received from acceptor a*
9. $Q_P \leftarrow Q_P \cup \{a\}$
10.   **if** $f \neq nil \wedge (e_{max} = nil \vee f > e_{max})$ **then**
11. $e_{max} \leftarrow f$, $v \leftarrow w$
12.   **case** *no-promise(e,f,g,w) received from acceptor a*
13. $\mathcal{E} \leftarrow \{n \in \mathcal{E} | n > f\}$
14.   **if** $(g, w) \neq nil \wedge f = g$ **then**
                    `/* copy proposal` (g, w) `and skip rest of phase one */`
15. $e \leftarrow g$, $v \leftarrow w$, $Q_A \leftarrow \{a\}$
16.   **goto** *line 21*
17.   **else**
18.   **goto** *line 1*
19. **if** $v = nil$ **then**
20. $v \leftarrow \gamma$
21. **send** *propose(e,v) to acceptors*
22. **while** $|Q_A| < \lfloor n_a/2 \rfloor + 1$ **do**
23.   **switch do**
24.   **case** *accept(e) or no-promise(_,_,e,v) or no-accept(_,_,e,v) received from acceptor a*
25. $Q_A \leftarrow Q_A \cup \{a\}$
26.   **case** *no-accept(e,f,g,w) received from acceptor a*
27. $\mathcal{E} \leftarrow \{n \in \mathcal{E} | n > f\}$
28.   **if** $(g, w) \neq nil \wedge f = g$ **then**
                    `/* copy proposal` (g, w) `and restart phase two */`
29. $e \leftarrow g$, $v \leftarrow w$, $Q_A \leftarrow \{a\}$
30.   **goto** *line 21*
31.   **else**
32.   **goto** *line 1*
33. **return** v

Example: Efficient recovery from NACKs

Earlier (in §3.1) we learned that negative responses (NACKs) could be used to provide proposers with additional information about the state of the acceptors. In Algorithm 5, the proposer restarted the proposer algorithm if a *no-promise(e,f)* or *no-accept(e,f)* was received. We also discussed that the NACKs could also include the last accepted proposal (g, w), for example *no-promise(e,f,g,w)* or *no-accept(e,f,g,w)*, but at this time this additional information was of no benefit.

Proposal copying allows proposers to utilise this information. If a negative response includes a proposal (g, w) that is not *nil*, then the proposer has learned that epoch g maps to value w. Instead of retrying the proposal from line 1, the proposer may jump to phase two of the proposal (g, w). This is shown in lines 14-16 and 28-30 of algorithm 11[14].

Furthermore, from negative responses to previous proposals which include the proposal (e, v), the proposer learns that the proposal (e, v) has been accepted by the acceptor thus this acceptor can count towards the phase two quorum (line 24, Algorithm 11).

Figure 3.5 shows a revised version of Figure 3.1 where proposer p_1 copies the proposal $(5, B)$ and thus skips phase one of epoch 5. For simplicity, this scenario assumes that the *no-promise* from acceptor a_2 is lost or delayed. As usual the proposer sent their requests to all acceptors, however in this case there is no need for proposer p_1 to send *propose(5,B)* to acceptor a_1 as p_1 already knows that a_1 has accepted the proposal $(5, B)$.

Example: Efficient recovery on co-located systems

Consider a system where proposers and acceptors are co-located on each participant and phase three is used for termination. If a participant has accepted a proposal (e, v) but not learned that a decision has been reached within a timeout, it could start a recovery proposer. Using proposal copying, the participant can skip phase one and proceed to phase two where it sends *propose(e,v)* to all other participants. Not only may the participant decide the value in only a single phase (phase two) but also there will be no conflict between the original proposer and the copying proposers.

In summary to copy the proposal (e, v), a participant first must learn that at some point *propose(e,v)* had been dispatched. This means that the value v has been chosen to correspond with the epoch e. In the next section, we explore what happens if instead values are statically pre-allocated to epochs.

[13] This is a generalisation of phase two bypass (§3.2).

[14] Algorithm 11 also includes the requirement that $f = g$ otherwise proposal copying is unlikely to be successful.

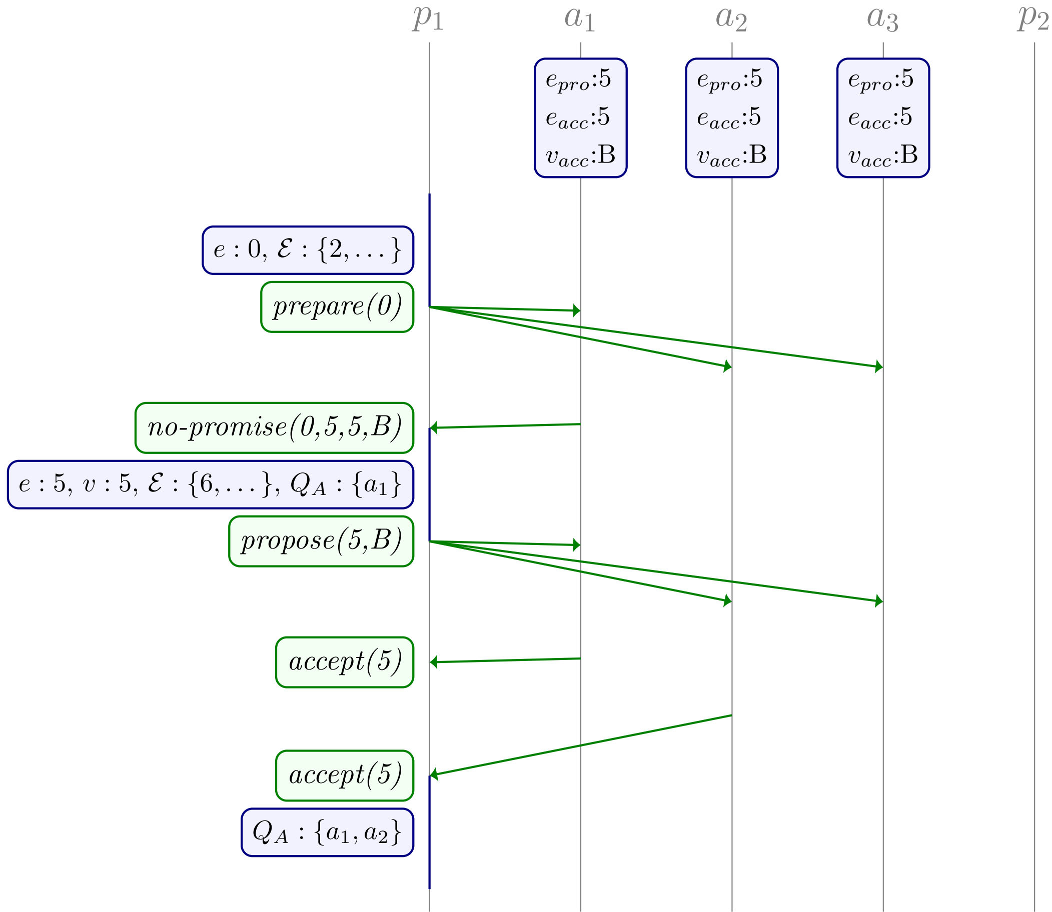

Figure 3.5: Classic Paxos with proposal copying from NACKs (Alg. 4,11)

3.11 Generalisation to quorums

Recall that we assume a finite set of acceptors $A = \{a_1, a_2, \dots, a_{n_a}\}$, with $|A| = n_a$.

Definition 6. *A quorum Q is defined as a non-empty subset of acceptors, $Q \in \mathcal{P}(A) \setminus \emptyset$.*

Definition 7. *A quorum set $\mathcal{Q}$ is a non-empty set of quorums, $\mathcal{Q} \subseteq \mathcal{P}(A) \setminus \emptyset$.*

Classic Paxos as described thus far uses *strict majority quorums*. Formally we define the quorum set as follows:

$$\mathcal{Q} = \{Q \in \mathcal{P}(A) | \ |Q| \geq \lfloor n_a/2 \rfloor + 1\}$$

Classic Paxos cannot make progress without majority participation, thus it is able to handle up to a minority $\lceil n_a/2 \rceil - 1$ of acceptors failing. This approach tightly couples the total number of acceptors, the number of acceptors needed to participate in consensus and the number of failures tolerated. Ideally, we would like to minimise the number of acceptors in the system and the number required to participate in consensus, as the proposer must wait upon the acceptors to reply and send more messages. Conversely, we would like to

maximise the number of failures handled by the systems. When using majorities, to handle f failures, the quorums must be of at least size $f+1$ in a system of $2f+1$ acceptors. This approach quickly limits the scalability and fault tolerance of a system.

The purpose of using strict majorities is to ensure that all quorums intersect, therefore it has been noted elsewhere [Lam78a, §1.4][Lam01a, §2.2][Lam05a][JRS11, §2] that majorities can be generalised to use any quorum system $\mathcal{Q}$, provided that all quorums $Q \in \mathcal{Q}$ intersect.

Therefore, we revise our definition of *decided* as follows:

Definition 8. *A proposal* (e, v) *is decided if the proposal* (e, v) *has been accepted by a quorum of acceptors.*

Formally, the *quorum intersection requirement* for Classic Paxos is specified as follows:

$$\forall Q, Q' \in \mathcal{Q} : Q \cap Q' \neq \emptyset \tag{3.1}$$

A generalised version of the proposer algorithm is given by Algorithm 12. The acceptor algorithm remains unchanged.

Figure 3.6 gives an example of this generalisation in practice. In this scenario, the system is comprised of 4 acceptors, $A = \{a_1, a_2, a_3, a_4\}$ and the quorum system is $\mathcal{Q} = \{\{a_1, a_2\}, \{a_1, a_3\}, \{a_1, a_4\}, \{a_2, a_3, a_4\}\}$. Compared to strict majority quorums, which would require three acceptor to form a quorum, the proposer p_1 in this example is able to complete both phases using a quorum of only two acceptors, a_1 and a_2[15].

Strict majorities are just one example of a quorum set which satisfies Classic Paxos's quorum intersection requirement. There are many quorum sets that could be utilised with Classic Paxos, offering different tradeoffs in sizes of quorums, number and diversity of quorums, number of participants as well as number and types of failures tolerated. The flexibility to choose a quorum set allows us to loosen the coupling between the number of acceptors, the number of acceptors participating in each phase and the number of failures tolerated. However, since all quorums are required to intersect, there remain fundamental limitations on what can be achieved. For example, Classic Paxos cannot reach a decision whilst any whole quorum has failed. As such, quorum systems other than strict majority are rarely utilised in practice.

3.12 Miscellaneous

Other variants and optimisations include:

[15]Equally, acceptors a_1 and a_3 or acceptors a_1 and a_4 could have be used. Different quorums may be used by each phase.

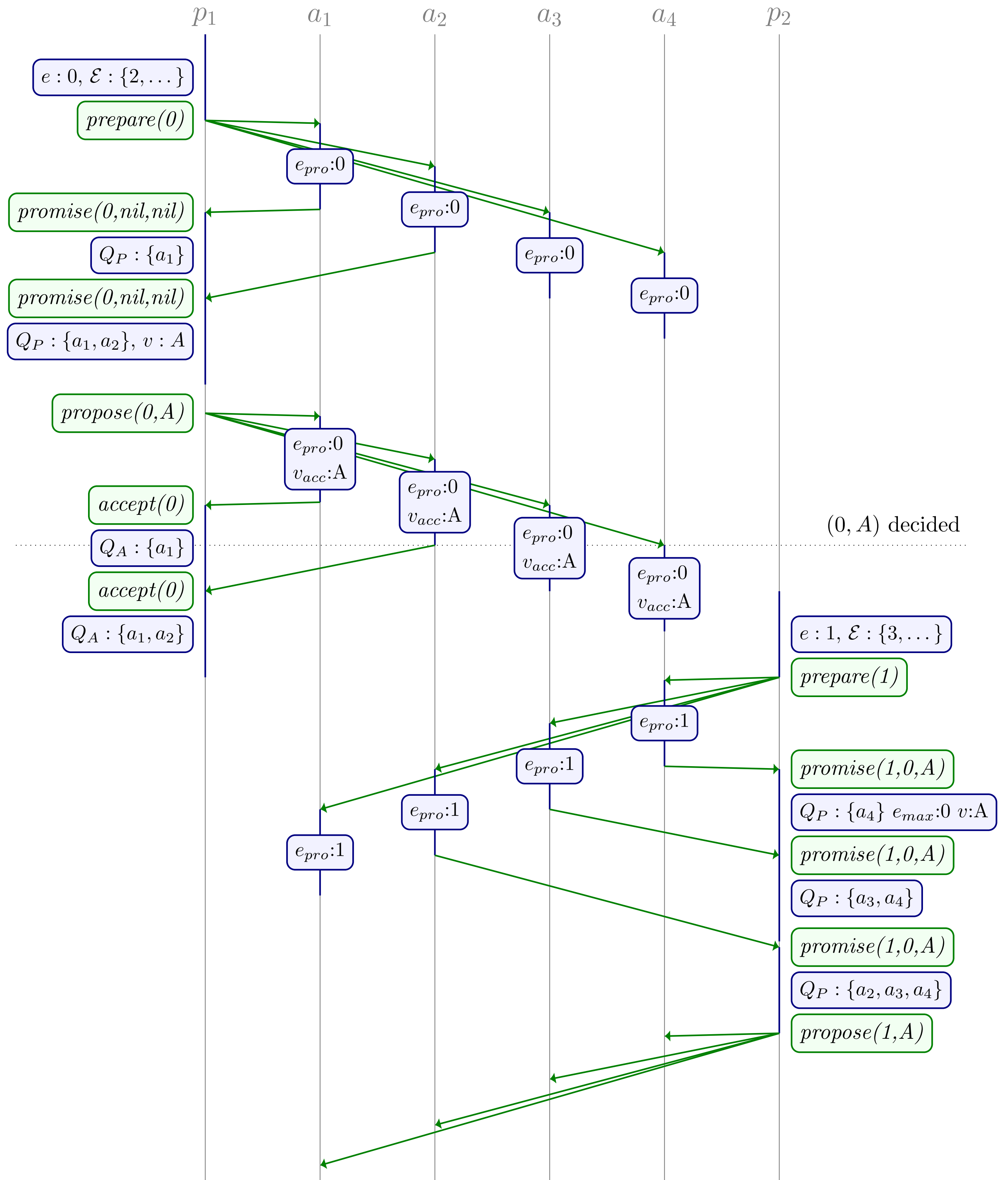

Figure 3.6: Classic Paxos with non-majority quorums (Alg. 4,12)

Algorithm 12: Proposer algorithm for Classic Paxos with generalised quorums

state:

- $\mathcal{Q}$: set of quorums (configured, persistent)

```
1  v, e_max ← nil
2  Q_P, Q_A ← ∅
3  e ← min(ℰ)
4  ℰ ← ℰ \ {e}
   /* Start Phase 1 for epoch e */
5  send prepare(e) to acceptors
6  while ∀Q ∈ 𝒬 : Q_P ⊉ Q do
7      switch do
8          case promise(e,f,w) received from acceptor a
9              Q_P ← Q_P ∪ {a}
10             if e_max = nil ∨ f > e_max then
11                 e_max ← f, v ← w
12         case timeout
13             goto line 1
14 if v = nil then
15     v ← γ
   /* Start Phase 2 for proposal (e,v) */
16 send propose(e,v) to acceptors
17 while ∀Q ∈ 𝒬 : Q_A ⊉ Q do
18     switch do
19         case accept(e) received from acceptor a
20             Q_A ← Q_A ∪ {a}
21         case timeout
22             goto line 1
23 return v
```

Learning

When discussing Paxos, a third role, referred to as a learner, is often considered. The learner is simply a participant wishing to learn the decided value. A learner is similar to a proposer, in that they wish to learn. Unlike a proposer, they are passive and do not have a value of their own to propose. Learners could be notified by either the proposers or the acceptors, once they learn that a value has been decided and what the decided value is. More options for learning are discussed elsewhere [Lam01a, §2.3]

Messages

In this chapter so far, proposers in Classic Paxos sent their prepare and propose messages to all n_a acceptors and waited for a majority to respond. If all acceptors are up then this approach generates $2n_a$ messages per phase. This approach is taken by systems such as Chubby [Bur06], VRR [LC12, §4.1], Raft [OO14] and Moderately Complex Paxos [VRA15].

Since proposers need only a majority of acceptors to respond, they can safely just send messages to a majority of acceptors and send further messages only if needed, for example if one or more acceptors do not reply. In the best case, when all acceptors are up, this method generates $2(\lfloor n_a/2 \rfloor + 1)$ messages. If we wish to reduce the likelihood of needing to send further messages, we can send more than a majority in the first place. This is the approach taken in Ring Paxos [MPSP10, §4].

If we wish to minimise the number of messages further, we can have acceptors forward messages in a chain or ring. In the best case, this approach reduces messages to $\lfloor n_a/2 \rfloor + 2$, however latency increases from 2 hops to $\lfloor n_a/2 \rfloor + 2$ hops. This is similar to the approach taken in phase two of Ring Paxos [MPSP10, §3].

Stricter epoch conditions

The Classic Paxos acceptor algorithm as described will promise/accept if a prepare/propose message has an epoch e greater than or equal to the last promised epoch e_{pro}. Some algorithms have stricter requirements such as [MPSP10, §4] which requires that $e > e_{pro}$ to promise and Moderately Complex Paxos [VRA15] which requires both that $e > e_{pro}$ to promise and $e = e_{pro}$ to accept. These restrictions are always safe, as they are equivalent to dropping a message but may effect the liveness conditions for progress.

Fail-stop model

We could avoid writing to persistent storage by not permitting participants to restart after a failure. This would however mean that number of participants decreases over time and the system would need reconfiguration to maintain its fault tolerance. This is the approach taken by VRR [LC12, §4.3]. Alternatively, we could require that no more than a majority of acceptors fail [MPSP10, §4.2].

Virtual sequences

When reaching consensus on a value as part of a sequence, it is useful to note that there is not necessarily a 1-to-1 correspondence between the values in the sequence and indexes used by the application. We can improve this by deciding at each index a sequence of values instead of a single value. This batching of values into decisions reduces decision latency

and need not be exposed externally as values can be re-assigned consequence (*virtual*) indexes. Batching is used extensively in consensus, examples include Chubby [CGR07], Mencius [MJM08], VRR [LC12, §6.2] and Raft [OO14]. This abstraction means that a sequence of length zero is a nil value which can be decided. We see *no-ops* like this utilised in various algorithms such as Simple Paxos [Lam01a, §3] and Mencius [MJM08].

Fast Paxos

Fast Paxos [Lam05a] is a variant of Classic Paxos whereby for a subset of epochs, if an acceptor receives no proposals within its phase one (and thus could propose its own value) then it can notify all other acceptors and any acceptor can propose their own value in phase two directly, without executing phase one again. The literature refers to these epochs as *fast* and all other epochs as *classic*. In addition to requiring that all quorums intersect, to preserve safety, Fast Paxos requires that any two fast and a classic quorum must intersect. Fast Paxos uses counting quorum of size k_f for fast epochs and k_c for classic epochs such that[16]:

$$n_a < 2k_c$$

$$2n_a < 2k_f + k_c$$

3.13 Summary

Classic Paxos has been the subject of extensive study and this chapter only begins to discuss the wide range of consensus algorithms within the Classic Paxos family. All algorithms in this family share three key characteristics: epochs, two phases and majority (or intersecting quorum) agreement. Over the next three chapters, we will revise each of these aspects, beginning with quorum intersection.

[16]These expressions are re-arranged from §3.4.1 of [Lam05a]

Chapter 4

Quorum intersection revised

In this chapter, we prove that the usual description of Classic Paxos (as given in Chapter 2) is more conservative than is necessary. More specifically, we will demonstrate that the quorum intersection requirement for Classic Paxos, which requires all quorums to intersect (formally stated by eq. 3.1 in §3.11), can be substantially weakened. This result has wide-ranging implications, which will be explored throughout this thesis. In particular, we will demonstrate that it provides much greater flexibility in how we reach distributed consensus.

This chapter progressively refines the quorum intersection requirements in two distinct stages: revision A (§4.1) & revision B (§4.2). We begin with the Classic Paxos generalisation to quorums (§3.11). Each stage generalises over the previous revision by further weakening the quorum intersection requirements.

4.1 Quorum intersection across phases

We begin by differentiating between the quorums used by phase one of Classic Paxos, which we will refer to as $\mathcal{Q}_1$ and the quorums for phase two, referred to as $\mathcal{Q}_2$. We could use different quorum sets of each of the two phases of Classic Paxos.

As before, we begin by revising our definition of *decided*:

Definition 9. *A proposal* (e, v) *is decided if the proposal* (e, v) *has been accepted by a phase two quorum of acceptors.*

Since Classic Paxos requires that all quorums intersect, regardless of the phase of the algorithm, the quorum sets $\mathcal{Q}_1, \mathcal{Q}_2$ must satisfy all the following three intersection requirements:

$$\forall Q, Q' \in \mathcal{Q}_1 : Q \cap Q' \neq \emptyset \tag{4.1}$$

$$\forall Q, Q' \in \mathcal{Q}_2 : Q \cap Q' \neq \emptyset \tag{4.2}$$

$$\forall Q_1 \in \mathcal{Q}_1, \forall Q_2 \in \mathcal{Q}_2 : Q_1 \cap Q_2 \neq \emptyset \tag{4.3}$$

Our first finding is that it is only necessary for phase one quorums ($\mathcal{Q}_1$) and phase two quorums ($\mathcal{Q}_2$) to intersect. There is no need to require that phase one quorums intersect with each other nor that phase two quorums intersect with each other. Since no intersection is required within phases, quorums within each phase of Classic Paxos can be disjoint. We will refer to this generalisation of Classic Paxos as Paxos revision A. In the literature, we referred to this as Flexible Paxos (FPaxos).

Formally, the *revision A quorum intersection requirement* can be stated as:

$$\forall Q_1 \in \mathcal{Q}_1, \forall Q_2 \in \mathcal{Q}_2 : Q_1 \cap Q_2 \neq \emptyset \tag{4.4}$$

4.1.1 Algorithm

Algorithm 13 gives the generalised pseudo-code for Classic Paxos. Only the proposer algorithm is provided here as the acceptor algorithm is unchanged from Algorithm 4. We could configure the algorithm with the quorums sets $\mathcal{Q}_1$ and $\mathcal{Q}_2$, or we can provide one of the quorums sets and calculate the other as needed. The latter is the approach taken in Algorithm 13.

4.1.2 Safety

We will now consider why it is safe to relax the quorum intersection requirement by examining how the intersection of quorums was utilised in the earlier safety proof for Classic Paxos (§2.6).

Recall the following properties (originally defined §2.4):

Property 2. *Proposers only propose a value after receiving promises from* $\lfloor n_a/2 \rfloor + 1$ *acceptors.*

Property 3. *Proposers only return a value after receiving accepts from* $\lfloor n_a/2 \rfloor + 1$ *acceptors.*

We will now replace them with the following properties for Paxos revision A. All other properties remain unchanged.

Property 11. *Proposers only propose a value after receiving promises from a phase one quorum of acceptors,* $Q \in \mathcal{Q}_1$.

Property 12. *Proposers only return a value after receiving accepts from a phase two quorum of acceptors,* $Q \in \mathcal{Q}_2$.

Algorithm 13: Proposer algorithm for Paxos revision A

```
state:
  • Q2: set of quorums for phase two (configured, persistent)

1  v, e_max ← nil
2  Q_P, Q_A ← ∅
3  e ← min(ℰ)
4  ℰ ← ℰ \ {e}
   /* Start Phase 1 for epoch e */
5  send prepare(e) to acceptors
6  while ∃Q ∈ Q2 : Q_P ∩ Q = ∅ do
7      switch do
8          case promise(e,f,w) received from acceptor a
9              Q_P ← Q_P ∩ {a}
10             if f ≠ nil ∧ (e_max = nil ∨ f > e_max) then
11                 e_max ← f, v ← w
12         case timeout
13             goto line 1
14 if v = nil then
15     v ← γ
   /* Start Phase 2 for proposal (e,v) */
16 send propose(e,v) to acceptors
17 while ∀Q ∈ Q2 : Q_A ⊉ Q do
18     switch do
19         case accept(e) received from acceptor a
20             Q_A ← Q_A ∪ {a}
21         case timeout
22             goto line 1
23 return v
```

From Table 2.3, we learn revising our proof of Lemma 11 is sufficient to prove the safety Paxos revisions A.

Recall Lemma 11 from our safety proof for Classic Paxos (§2.6):

Lemma 11 (Quorum intersection)**.** *If a value v is decided in epoch e then at least one acceptor which accepted proposal (e, v) will be required to promise in any future proposals $> e$.*

Lemma 11 proved that at least one acceptor which accepted a decided proposal will be required to promise in any subsequent proposals. This was trivially proven by Classic Paxos's requirement that a quorum of acceptors participate in each phase of the algorithm

and by the requirement that any two quorums will intersect. We can however also prove lemma 11 using the weaker revision A quorum intersection from equation 4.4:

Revised proof of lemma 11. Assume the value v is decided in epoch e, thus some phase two quorum of acceptors $Q_2 \in \mathcal{Q}_2$ would have accepted the proposal (e, v).

Before a value is proposed in phase two, a phase one quorum $Q_1 \in \mathcal{Q}_1$ of acceptors must promise to the proposer (Property 11). From equation 4.4, these two quorums will always intersect therefore the quorums will always have at least one acceptor in common. □

The proof of lemma 11 was the only occasion that quorum intersection was utilised in the proof of Classic Paxos. Therefore, we can substitute the above into the original proof of Classic Paxos, for proof of safety for Paxos revision A. For the sake of brevity, we do not reproduce the full proof here. The proof of non-triviality for Classic Paxos (§2.5) did not utilise quorum intersection and therefore still applies for Paxos Revisions A.

4.1.3 Examples

Figures 4.1 and 4.2 illustrate two example executions of Paxos revision A. In both cases, the system is comprised of four acceptors $A = \{a_1, a_2, a_3, a_4\}$ and two proposers $P = \{p_1, p_2\}$. The quorum system is as follows: $\mathcal{Q}_1 = \{\{a_1, a_2\}, \{a_3, a_4\}\}$ and $\mathcal{Q}_2 = \{\{a_1, a_3\}, \{a_2, a_4\}\}$. This quorum system has been chosen as it has the minimum intersection to satisfy the revised quorum intersection requirements. For simplicity, the acceptors in this example only send messages to one quorum for each phase instead of all possible quorums. Figure 4.1 shows the two proposers executing Paxos revision A in serial. Proposer p_1 decides the proposal $(0, A)$ prior to proposer p_2 starting the proposer algorithm. As expected, the proposer p_2 decides the proposal $(1, A)$. Figure 4.2 shows the two proposers executing Paxos revision A concurrently. Both proposers are able to complete phase one as the two phase one quorums used are disjoint. However, only p_2 is able to complete phase two due to the intersection between p_1's phase two quorum and p_2's phase one quorum at acceptor a_3. Proposer p_1 subsequently retries with epoch 2 and $(2, B)$ is decided.

4.2 Quorum intersection across epochs

In the previous section, we differentiated between the quorums for each phase of Paxos. We continue this refinement by differentiating between the quorums by their associated epochs, $e \in E$ as well as their phase. We use Q^e_n to denote the quorum set for phase n with epoch e. Thus far we have used the same quorum set regardless of the epoch. If we used epoch specific quorums sets, we would require the following for each epoch e:

$$\forall Q \in \mathcal{Q}^e_1, \forall f \in E, \forall Q' \in \mathcal{Q}^f_2 : Q \cap Q' \neq \emptyset \tag{4.5}$$

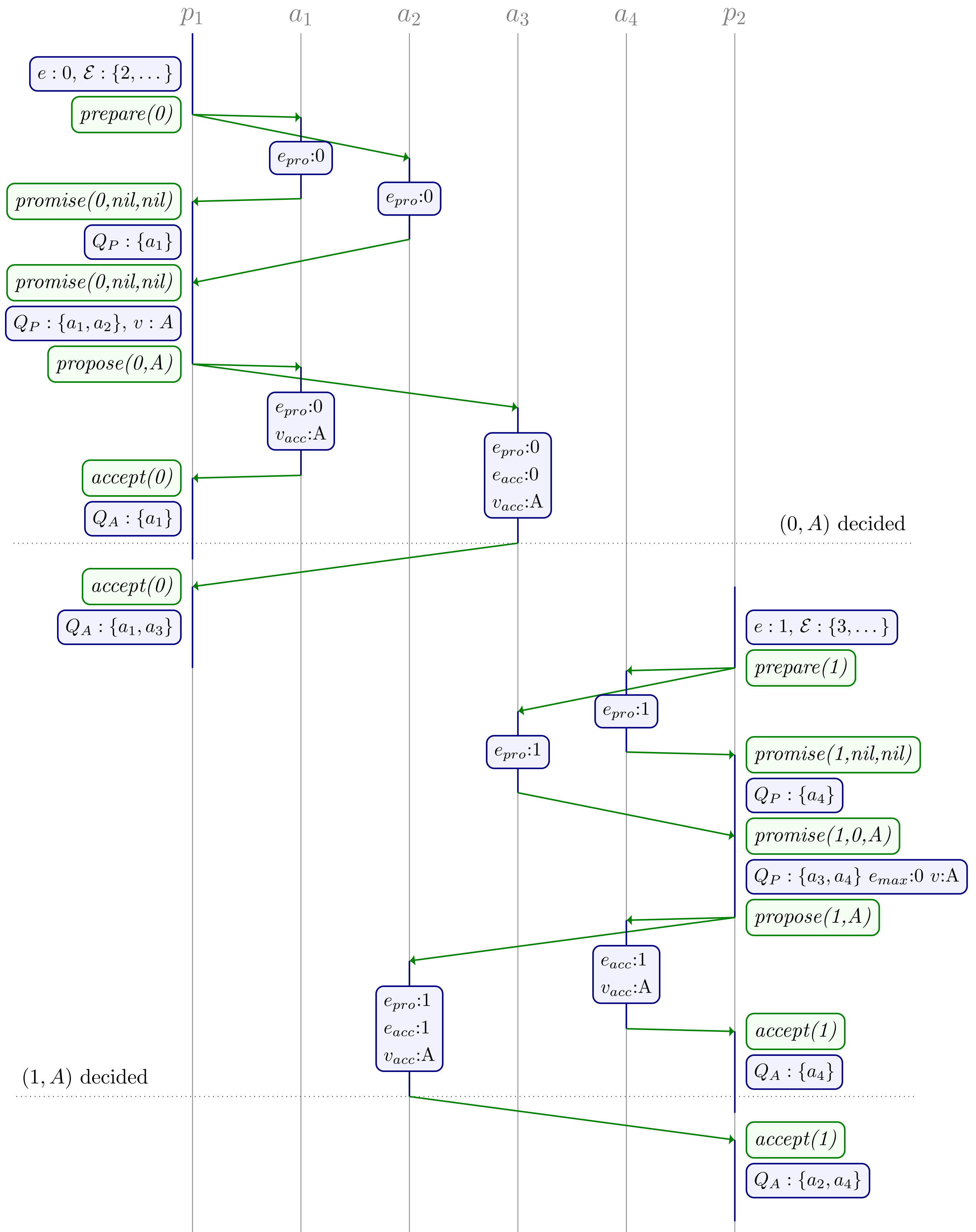

Figure 4.1: Example run of Paxos revision A with disjoint quorums within each phase and two serial proposers.

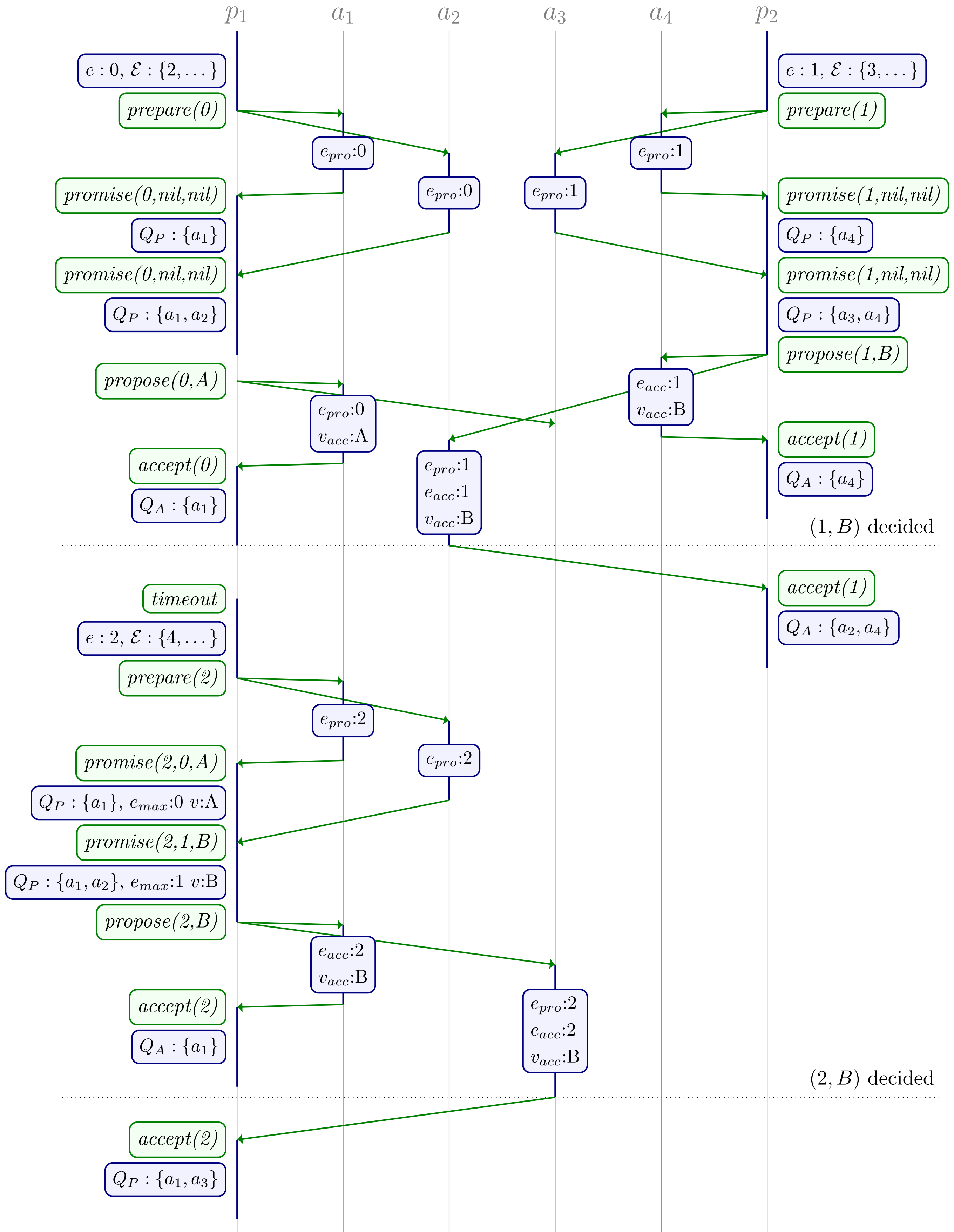

Figure 4.2: Example run of Paxos revision A with disjoint quorums within each phase and two concurrent proposers.

As before, we begin by revising our definition of *decided*:

Definition 10. *A proposal* (e, v) *is decided if the proposal* (e, v) *has been accepted by a phase two quorum of acceptors for epoch* e*.*

Our next result observes that we can further weaken the quorum intersection requirement. We require only that a phase one quorum of the epoch e ($\mathcal{Q}_1^e$) intersects with the phase two quorums ($\mathcal{Q}_2^f$) for all smaller epochs, $\{f \in E | f < e\}$[1]. There is no requirement that the acceptors for the phase one and phase two quorums for a given epoch intersect. Likewise, there is no requirement that a phase one quorum of the epoch must intersect with the phase two quorums for all greater epochs.

This newly revised quorums intersection requirement, referred to as the *revision B quorum intersection requirement*, can be specified as follows for each epoch e:

$$\forall Q \in \mathcal{Q}_1^e, \forall f \in E : f < e \implies \forall Q' \in \mathcal{Q}_2^f : Q \cap Q' \neq \emptyset \tag{4.6}$$

4.2.1 Algorithm

Algorithm 14 gives the revised generalised pseudo-code for Classic Paxos. Only the proposer algorithm is provided here as the acceptor algorithm is unchanged from Algorithm 4. Note that it is now possible for a phase one quorum to be empty (as we will discuss later) thus we can add the option to skip phase one in this case.

4.2.2 Safety

Similar to the case of revision A (Equation 4.4), the safety of this result is derived from the observation that the proof of safety for Classic Paxos does not use the full strength of the assumptions made regarding quorum intersection.

Recall the following properties (originally defined §2.4):

Property 2. *Proposers only propose a value after receiving promises from* $\lfloor n_a/2 \rfloor + 1$ *acceptors.*

Property 3. *Proposers only return a value after receiving accepts from* $\lfloor n_a/2 \rfloor + 1$ *acceptors.*

As before, we begin by redefining Properties 2 & 3. All other properties remain unchanged.

[1]Or equivalently, that a phase two quorum of an epoch e intersects with the phase one quorums from all greater epochs.

Algorithm 14: Proposer algorithm for Paxos revision B

state:
- $\mathcal{Q}_2^e$: for each $e \in E$, set of quorums for phase two (configured, persistent)

```
1  v, e_max ← nil
2  Q_P, Q_A ← ∅
3  e ← min(ℰ)
4  ℰ ← ℰ \ {e}
   /* Start Phase 1 for epoch e */
5  send prepare(e) to acceptors
6  while ∃z ∈ E : z < e ∧ ∃Q ∈ Q_2^z : Q_P ∩ Q = ∅ do
7      switch do
8          case promise(e,f,w) received from acceptor a
9              Q_P ← Q_P ∩ {a}
10             if f ≠ nil ∧ (e_max = nil ∨ f > e_max) then
11                 e_max ← f, v ← w
12         case timeout
13             goto line 1
14 if v = nil then
15     v ← γ
   /* Start Phase 2 for proposal (e,v) */
16 send propose(e,v) to acceptors
17 while ∀Q ∈ Q_2^e : Q_A ⊉ Q do
18     switch do
19         case accept(e) received from acceptor a
20             Q_A ← Q_A ∪ {a}
21         case timeout
22             goto line 1
23 return v
```

Property 13. *Proposers only propose a value in epoch e after receiving promises from a phase one quorum of acceptors for epoch e, $\mathcal{Q}_1^e$.*

Property 14. *Proposers only return a value after receiving accepts from a phase two quorum of acceptors for epoch e, $\mathcal{Q}_2^e$.*

Recall Lemma 11 from our safety proof for Classic Paxos (§2.6):

Lemma 11 (Quorum intersection)**.** *If a value v is decided in epoch e then at least one acceptor which accepted proposal (e, v) will be required to promise in any future proposals $> e$.*

Similar to the safety proof of Paxos revision A (§4.1.2), we now provide a revised proof of lemma 11 which can be substituted into the proof of Classic Paxos for a proof of safety for Paxos revision B.

Revised proof of lemma 11. Assume the value v is decided in epoch e, thus some phase two quorum of acceptors $Q \in \mathcal{Q}_2^e$ would have accepted the proposal (e, v).

Consider a proposal in epoch f where $f > e$. Before a value could be proposed in f, a phase one quorum of acceptors for proposal in epoch f, $Q' \in \mathcal{Q}_1^f$ must promise to the proposer of f (Property 13). Since $f > e$, we can apply equation 4.6 to see that any two quorums will intersect therefore the quorums will always have at least one acceptor in common. □

Revision A generalises over Classic Paxos by weakening the quorum intersection requirements depending on the algorithm phase the quorum is used with. In turn, Revision B generalises over Revision A (and therefore Classic Paxos) by weakening the quorum intersection requirements depending on the epoch and phase that the quorum is used with. Like our proof of safety for Classic Paxos, Lamport's original proof did not use the full strength of the assumptions that were made, namely that all quorums will intersect. This result does not dispute that Classic Paxos is a solution to distributed consensus but does demonstrate that the algorithm is needlessly conservative in its approach. Classic Paxos is a specific case of Paxos revision A and in turn of, Revision B, which adds the requirement for quorum intersection within each phase and regardless of epoch.

4.2.3 Examples

A key implication of this result is that for the minimum epoch e_{min} where $e_{min} = min(E)$, there is no phase one quorum intersection requirement. The practical application of this is that a proposer with epoch e_{min} may skip phase one and proceed directly to proposing their own value γ in phase two using *propose*(e_{min},γ). As epochs are unique to proposers, only one proposer will be able to take advantage of this. Assuming this proposer is the first to propose a value and no other proposers try to propose concurrently, this proposer can decide a value in one round trip, as demonstrated in Figure 4.3. Figure 4.3 shows the same example as Figure 2.2, however the proposer p_1 is now able to skip phase one and reach agreement in just one phase.

This result is functionally equivalent to starting a system in a state such that one proposer has already executed phase one with all acceptors. This technique was utilised in the Coordinated Paxos algorithm in Mencius [MJM08, §4.2]

Counterintuitively, it is now possible that the commit point may have already been reached and that a proposer (with a lower epoch such as e_{min}) does not see the chosen value during

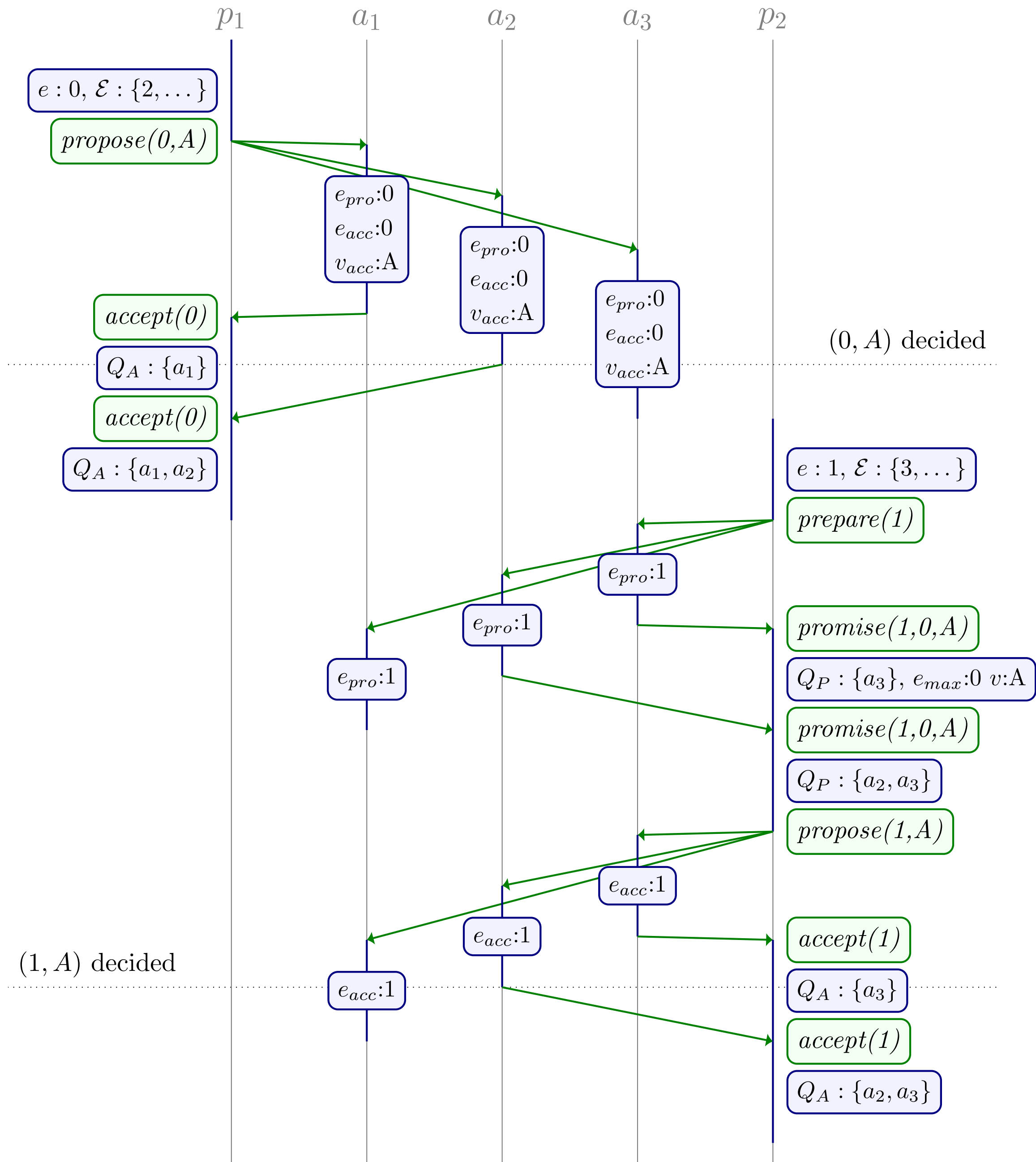

Figure 4.3: Example of a proposer successfully skipping phase one using the minimum epoch.

its phase one. This proposer may then propose a value, different to the decided value during its phase two. This situation does not cause a violation in safety since the proposer's phase two will be unsuccessful since the phase two quorum will intersect with the phase one of the higher epoch. An example of this case is shown in Figure 4.4. Figure 4.4 shows the same execution as Figure 2.3, however now the proposer p_1 skips over the first phase one.

More generally, the implication of this result is that phase one quorums are required only to intersect with the phase two quorums of previous epochs, instead of all phase two quorums. One application of this result is that if we vary phase two quorums with epochs

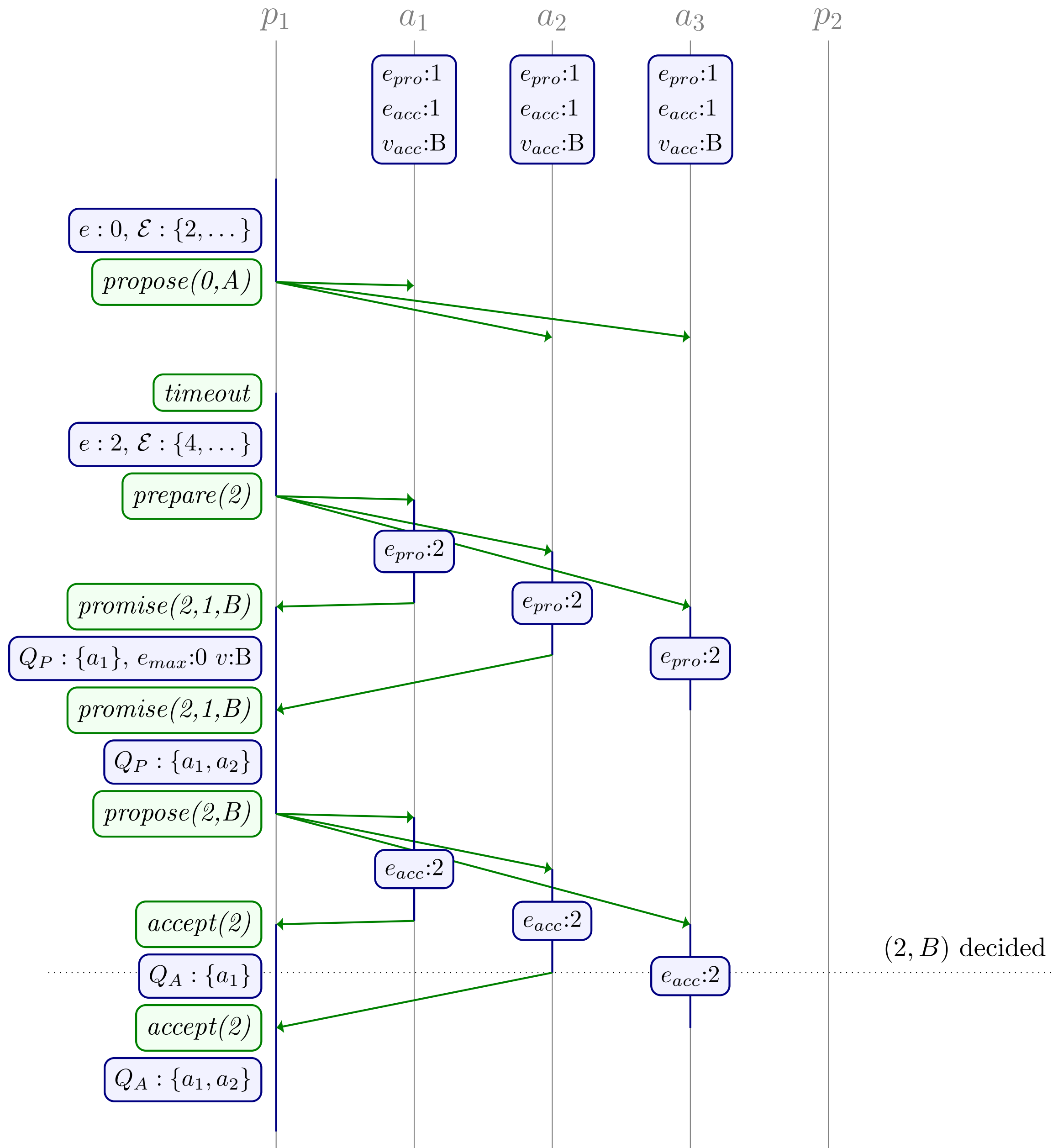

Figure 4.4: Example of a proposer proposing a value different to the decided value, after the commit point has been reached.

then we can reduce the phase one quorum depending on the epoch.

4.3 Implications

Thus far we have weakened the quorum intersection requirement of Paxos and discussed the implications for Classic Paxos, as described in Chapter 2. In this section, we will explore the implication of our revised understanding of consensus on the known variants of Paxos, as surveyed in Chapter 3.

4.3.1 Bypassing phase two

In section 3.2, we discuss how a Classic Paxos proposer can bypass phase two when a majority of acceptors return the same proposal (e, v) with promises in phase one. This is safe as (e, v) has already has been decided. The analogous optimisation is to return the value v when a phase two quorum $\mathcal{Q}_2^e$ of acceptors return the proposal (e, v). This can result in not only skipping phase two but also skipping the remainder of phase one, if a $\mathcal{Q}_2^e$ of acceptors return the same proposal before a $\mathcal{Q}_1^f$ of acceptors return promises.

4.3.2 Co-location of proposers and acceptors

In section 3.7, we discuss the option of co-locating both a proposer and an acceptor in each participant. We will now look at three algorithms which arise from combining this co-location with our weakened quorum intersection requirements.

Example: All aboard Paxos

One interesting implication of revision A that if we are willing to require all participants be up for guaranteed progress (and co-locate proposers and acceptors) then we can reach consensus in only one round trip. This is achieved by requiring all acceptors to accept in phase two. It is then sufficient under revision A for any acceptor to promise in phase one, as the intersection between the phases is still guaranteed. By co-locating the acceptors and proposers phase one can be completed locally without any communication with other participants.

For example, in a system of 3 acceptors $A = \{a_1, a_2, a_3\}$, the following are valid quorum sets:

$$\mathcal{Q}_1 = \{\{a_1\}, \{a_2\}, \{a_3\}\}$$
$$\mathcal{Q}_2 = \{\{a_1, a_2, a_3\}\}$$

In contrast, under Classic Paxos we would still require intersecting quorums, such as majorities, for phase one so there is no advantage (only disadvantage) to requiring all acceptors to participate in phase two.

Thus far, we have utilised revision A to achieve one round trip consensus provided all acceptors participated in phase two. The primary limitation of All aboard Paxos compared to Classic Paxos is that all participants must be live to guarantee progress instead of just a majority. We will now utilise revision B to overcome this limitation as follows. We will require all acceptors to accept in phase two for the epochs 0 to some epoch k. We will only require majorities to accept in phase two for all epochs from $k+1$. Any value greater

	Phase one quorums, $\mathcal{Q}_1^e =$	Phase two quorums, $\mathcal{Q}_2^e =$
$e = 0$	$\{\{\}\}$	$\{\{a_1, a_2, a_3\}\}$
$e \in [1, k]$	$\{\{a_1\}, \{a_2\}, \{a_3\}\}$	$\{\{a_1, a_2, a_3\}\}$
$e = k + 1$	$\{\{a_1\}, \{a_2\}, \{a_3\}\}$	$\{\{a_1, a_2\}, \{a_2, a_3\}, \{a_1, a_3\}\}$
$e \in [k + 2, \infty]$	$\{\{a_1, a_2\}, \{a_2, a_3\}, \{a_1, a_3\}\}$	$\{\{a_1, a_2\}, \{a_2, a_3\}, \{a_1, a_3\}\}$

Table 4.1: Example quorums for All aboard Paxos with three acceptors $U = \{a_1, a_2, a_3\}$.

than or equal to 1 can be chosen for k. An example set of phase two quorums is shown in the third column of Table 4.1.

Without revision B, we would require majority quorums for all phase ones, regardless of the epoch, to ensure quorum intersection across phases. However, using the weakened quorum intersection requirements for revision B, we can reduce the phase one quorums. As we have already discussed, there is no phase one quorum intersection requirement for epoch 0. For proposals numbers 1 to $k + 1$, any acceptor is a valid phase one quorum. For epochs $k + 2$ onward, any majority of acceptors is a valid phase one quorum. The result is that proposers can fall back to Classic Paxos if they do not receive responses from all acceptors. An example set of phase one quorums is shown in the second column of Table 4.1.

A decision can be reached in one round trip if all acceptors are available (and provided no proposer has tried to propose in epoch $> k$) or two round trips if only a majority of acceptors are available.

Example: Singleton Paxos

Alternatively, we could instead require that all acceptors promise in phase one thus allowing any acceptor to accept a value in phase two. For example, in a system of 3 acceptors $A = \{a_1, a_2, a_3\}$, the following are also valid quorum sets:

$$\mathcal{Q}_1 = \{\{a_1, a_2, a_3\}\}$$
$$\mathcal{Q}_2 = \{\{a_1\}, \{a_2\}, \{a_3\}\}$$

The phase two could also include a phase three to store the decided value, as described in §3.3.

Example: Majority quorums for co-location

The idea of using different quorums for different epochs, as proposed in §4.2 may seen unusual, however this is already common place. Consider a Classic Paxos system of 5 participants $U = \{u_1, u_2, u_3, u_4, u_5\}$, where each participant is both an acceptor and proposer.

Epochs are pre-allocated in a round robin fashion such that participant u_1 may use epochs $\mathcal{E} = \{0, 5, 10, \dots\}$, participant u_2 may use epoch $\mathcal{E} = \{1, 6, 11, \dots\}$ and so on. We assume our system is using majority quorums, therefore regardless of phase or epoch, our quorums are as follows:

$$Q = \{\{u_1, u_2, u_3\}, \{u_1, u_2, u_4\}, \{u_1, u_2, u_5\}, \{u_1, u_3, u_4\}, \{u_1, u_3, u_5\},$$
$$\{u_1, u_4, u_5\}, \{u_2, u_3, u_4\}, \{u_2, u_3, u_5\}, \{u_2, u_4, u_5\}, \{u_3, u_4, u_5\}\}$$

In practice, however each participant will include itself in its quorums. Therefore the phase two quorums will be of the form:

$$Q_2^0 = \{\{\boldsymbol{u_1}, u_2, u_3\}, \{\boldsymbol{u_1}, u_2, u_4\}, \{\boldsymbol{u_1}, u_2, u_5\}, \{\boldsymbol{u_1}, u_3, u_4\}, \{\boldsymbol{u_1}, u_3, u_5\}, \{\boldsymbol{u_1}, u_4, u_5\}\}$$
$$Q_2^1 = \{\{u_1, \boldsymbol{u_2}, u_3\}, \{u_1, \boldsymbol{u_2}, u_4\}, \{u_1, \boldsymbol{u_2}, u_5\}, \{\boldsymbol{u_2}, u_3, u_4\}, \{\boldsymbol{u_2}, u_3, u_5\}, \{\boldsymbol{u_2}, u_4, u_5\}\}$$

The insight from revision B is that the phase one quorums need only intersect with the phase two quorums of smaller epochs. We are therefore able to refine the first few phase one quorums as follows:

$$Q_1^0 = \{\{\}\}$$
$$Q_1^1 = \{\{u_1\}, \{u_2, u_3, u_4\}, \{u_2, u_3, u_5\}, \{u_2, u_4, u_5\}, \{u_3, u_4, u_5\}\}$$
$$Q_1^2 = \{\{u_1, u_2\}, \{u_1, u_3, u_4\}, \{u_1, u_3, u_5\}, \{u_1, u_4, u_5\},$$
$$\{u_2, u_3, u_4\}, \{u_2, u_3, u_5\}, \{u_2, u_4, u_5\}, \{u_3, u_4, u_5\}\}$$

We could generalise this example across any quorum system to say that the set of all participants associated with previous epochs $< e$ is a valid phase one quorum for epoch e. In this specific example, the phase one quorums of the first three epochs have been improved, however, this insight it not helpful for epochs > 3 since any set of 3 or more participants is already a valid quorum. We address this in the next section.

4.3.3 Multi-Paxos

In Multi-Paxos (§3.6), the steady state of the algorithm is a proposer executing phase two with a majority of acceptors. If we assume that failures occur rarely, then phase one of Classic Paxos would be seldom executed compared to phase two. From Paxos revision A, we learned that quorum intersection is required only between phase one and phase two quorums. As a result, we can tradeoff between quorums sets for the phase one and phase

two. We can reduce the size (and/or increase the number) of phase two quorums at the cost of increasing the size (and/or reducing the number) of phase one quorums.

Multi-Paxos with majority quorums tightly couples performance, system size and fault tolerance. Systems may now choose the most suitable trade-offs for a given scenario. This modification optimises for the steady-state performance, whilst increasing the cost of recovering from failure. One exception to this rule is known as the *even nodes optimisation*. When the number of acceptors, n_a is even then the quorum size for Multi-Paxos is $\frac{n_a}{2}+1$ thus existing Multi-Paxos systems recommend against deploying on an even number of acceptors. With Paxos revision A, we can reduce the phase two quorum to $\frac{n_a}{2}$ for even n_a, making deployment on an even number of acceptors a viable option. This improvement to the phase two quorum size has no penalties elsewhere thus is effectively free.

The leader learns that a decision has been successfully reached once it receives accepts from a majority of acceptors. If we assume that propose messages are sent to all acceptors, the latency is therefore bounded by the round trip time to the fastest majority of acceptors. By reducing the size of the phase two quorum (and/or increasing the number of quorums), this latency is reduced (or, in the worst case, latency is unchanged). Reducing the decision latency thus increases the throughput which can be achieved under load[2]. Multi-Paxos is already widely deployed in practice. As such this optimisation to Multi-Paxos, even if marginal, can have wide-reaching impact with minimal implementation effort.

As previously discussed (§3.12), it is necessary only for the proposer to send propose messages to a phase two quorum of acceptors, provided the proposer can retry with another phase two quorum if an acceptor does not respond. This approach (almost) halves the number of messages sent per decision during the steady state of Classic Paxos, thus reducing the load on the leader and on the network. By having only the minimum number of acceptors accept each value, the overall storage requirement is also reduced. However, compared to sending propose messages to all acceptors, decision latency is increased both with and without failures. By reducing the size of the phase two quorum (and/or increasing the number of quorums), we can further reduce the number of messages and copies of accepted value.

One approach would be to alternate between groups in a set of disjoint quorums. This approach could vastly improve the throughput. This approach also reduces the space requirements for storing the sequence and is similar to sharding the sequence. An alternative approach would be to have a leader use a small fixed quorum of acceptors for the phase two. The remaining acceptors would be standbys since they are only needed in the case of failure.

[2]This is assuming the algorithm has some bound on the number of concurrent decisions and ignores the effects of batching decisions.

4.3.4 Voting for epochs

Previously, we discussed how Classic Paxos's phase one can be used to ensure uniqueness of epochs (§3.9). This observation can also apply to our revisions, provided phase one quorums for a given epoch intersect. This requires us to add the following quorum intersection requirement:

$$\forall Q, Q' \in \mathcal{Q}_1^e : Q \cap Q' \neq \emptyset \tag{4.7}$$

This mechanism allows any proposer to try to use any epoch, including e_{min}. However, this quorum intersection restriction means that we are no longer able to skip phase one for e_{min}.

4.4 Summary

Classic Paxos (§3.11) requires proposers to wait to complete phase one until they have received a promise from every quorum, regardless of the phase or epoch. In this chapter, we introduced revision A, proving that a proposer could complete phase one once it has received a promise for every phase two quorum, regardless of the epoch. Subsequently, we further weakened the Paxos intersection requirements in revision B, by proving that a proposer using epoch e can complete phase one once it has received a promise from each phase two quorum for epoch less than e.

Classic Paxos	$\exists Q \in \mathcal{Q} : Q_P \cap Q = \emptyset$
Revision A	$\exists Q \in \mathcal{Q}_2 : Q_P \cap Q = \emptyset$
Revision B	$\exists f \in E : f < e \wedge \exists Q \in \mathcal{Q}_2^f : Q_P \cap Q = \emptyset$

Table 4.2: Alternative phase one *while* conditions

Table 4.2 summaries how the three stages of generalisation have weakened the quorum intersection requirements. The expressions in Table 4.2 are alternative *while* conditions for completion of phase one.

Chapter 5

Promises revised

Classic Paxos (Chapter 2) requires proposers to wait until they have received promises from a majority of acceptors before proposing a value in phase two of the algorithm. In the last chapter (Chapter 4), we refined this to require proposers to wait until they have received promises from a phase two quorum of acceptors for each previous epoch before proceeding. Classic Paxos, and our revisions thus far, all require a proposer to wait for sufficient promises before proceeding, regardless of the content of the promises received[1].

In this chapter, we will demonstrate that the information learned from the promises received can be utilised to improve the flexibility of these algorithms. We will prove that proposers can safely proceed to phase two early depending on the content of the promises received in phase one.

5.1 Intuition

Paxos revision B requires that a proposer's phase one quorum must intersect with all possible phase two quorums for each previous epoch. This is because the proposer has no knowledge of which phase two quorums were used by other proposers. Consider what happens when a proposer receives *promise(e,f,v)* from an acceptor during phase one for epoch e. This proposer has learned that if a decision was reached in epoch f then the value chosen was v. This proposer need not wait for promises from all phase two quorums of f, $\mathcal{Q}_2^f$, as they will not return a promise with same epoch but a different value (Corollary 9.1).

Moreover, now that the proposer knows that value v was proposed in epoch f then the proposer does not need to intersect with phase two quorums associated with previous epochs $< f$.

[1]The exception to this statement is bypassing phase two when a majority of proposers promise with the same proposal (§3.2).

Specifically, if a proposer with epoch e learns the outcome of the value selection rule of epoch f, then the quorum intersection requirement for Paxos revision B can be reduced to:

$$\forall Q \in \mathcal{Q}_1^e, \forall g \in E : f < g < e \implies \forall Q' \in \mathcal{Q}_2^g : Q \cap Q' \neq \emptyset \tag{5.1}$$

This is known as the *revision C quorum intersection requirement.*

Recall that the purpose of phase one in Paxos is twofold: Firstly, to learn if a value may have already been decided and secondly, to prevent values from being decided between this phase and the next. If a proposer in e receives *promise(e,f,v)* then it learns that all epochs $\leq f$ are limited to value v. This is because the proposer of f must have received promises from a quorum of acceptors in its phase one. The proposer of e can essentially reuse the phase one that was successfully executed by the proposer in f as the outcome of phase one in epoch f is known to be value v.

5.2 Algorithm

Algorithm 15 gives the revised proposer algorithm for Paxos revision C. In contrast to the Paxos revision B (Algorithm 14), e_{max}, the greatest epoch received with a promise is used as an (exclusive) lower bound on the quorum intersection requirement for completing phase one (line 6, Algorithm 15).

5.3 Safety

We will prove the safety of Paxos revision C using the same approach that we adopted for our proof of safety for Classic Paxos (§2.6).

Recall the following property (originally defined §4.2.2):

Property 13. *Proposers only propose a value in epoch e after receiving promises from a phase one quorum of acceptors for epoch e, $\mathcal{Q}_1^e$.*

We revised Property 13 as follows and all other properties remain unchanged.

Property 15. *Proposers only propose a value in epoch e after receiving sufficient promises from the acceptors. For each previous epoch $f < e$, this is satisfied by either promises from at least one acceptor in each phase two quorum for f, $\mathcal{Q}_2^f$ or a promise from any acceptor including a proposal from epoch f or a subsequent epoch (*promise(e,g,_) *where $g \geq f$).*

Lemma 11 no longer holds. We begin by proving a weaker version of Lemma 11

Algorithm 15: Proposer algorithm for Paxos revision C.

```
1  v, e_max ← nil
2  Q_P, Q_A ← ∅
3  e ← min(ℰ)
4  ℰ ← ℰ \ {e}
   /* Start Phase 1 for epoch e  */
5  send prepare(e) to acceptors
6  while ∃z ∈ E, ∃Q ∈ 𝒬₂^z : (e_max = nil ∨ e_max < z) ∧ z < e ∧ (Q_P ∩ Q = ∅) do
7      switch do
8          case promise(e,f,w) received from acceptor a
9              Q_P ← Q_P ∩ {a}
10             if f ≠ nil ∧ (e_max = nil ∨ f > e_max) then
11                 e_max ← f, v ← w
12         case timeout
13             goto line 1
14 if v = nil then
15     v ← γ
   /* Start Phase 2 for proposal (e,v) */
16 send propose(e,v) to acceptors
17 while ∀Q ∈ 𝒬₂^e : Q_A ⊉ Q do
18     switch do
19         case accept(e) received from acceptor a
20             Q_A ← Q_A ∪ {a}
21         case timeout
22             goto line 1
23 return v
```

Lemma 18 (Weakened quorum intersection for Paxos revision C)**.** *If a value v is decided in epoch e then in all subsequent epochs either:*

- *at least one acceptor which accepted proposal (e, v) will promise, or*
- *an acceptor will promise with the proposal (e, v) or a proposal from a subsequent epoch.*

Proof of lemma 18. Assume the value v is decided in epoch e, thus some phase two quorum of acceptors $Q \in \mathcal{Q}_2^e$ would have accepted the proposal (e, v).

Consider a proposal in epoch f where $f > e$. Before a value could be proposed in f, a phase one quorum of acceptors for proposal in epoch f must promise to the proposer of f (Property 15). □

Thus we must provide a revised proof for Lemma 12 to verify the safety of Paxos revision C.

Lemma 12 (Weakened safety of future proposals)**.** *If a value v is decided in epoch e and value w is proposed in f where $f > e$ then w must have been proposed in g where $e \leq g < f$*

Revised proof of lemma 12. Assume value v is decided in epoch e and value w is proposed in f where $f > e$.

The proposer in f will have proposed w after completing phase one and choosing w as a result of the value selection rules.

From theorem 18, we know that either at least one acceptor which accepted proposal (e, v) will be required to promise in f or an acceptor will promise with a proposal from epoch e or a subsequent epoch.

In either case, the acceptor will reply with *promise(f,g,x)* where $e \leq g < f$ and x is the value proposed in g (Lemmas 6 & 10, Corollary 8.1).

According to the value selection rules (Property 4), the proposer of f must therefore propose either the value x or another value y from the proposal (h, y) such that $h > g$. Regardless of whether $w = x$ or $w = y$, w must have been proposed in an epoch between e (inclusive) and f (exclusive). □

The proof of non-triviality for Classic Paxos (§2.5) still applies to Paxos revision C.

5.4 Examples

The implications of this result apply even when the quorum system used is agnostic to the epoch. For example, we can extend the proposer algorithm for Classic Paxos to test whether a promise message includes a proposal for the predecessor of the current epoch. If this is the case, the proposer can proceed directly to phase two without waiting for a phase one quorum. Table 5.1 shows how line 6 of Algorithm 15 can be simplified if phase two quorums are epoch agnostic.

Revision A	$\exists Q \in Q_2 : Q_P \cap Q = \emptyset$
Revision B	same as A and $e \neq e_{min}$
Revision C	same as B and $e \neq succ(e_{max})$

Table 5.1: Simplified *while* conditions for line 6, Algorithm 15.

Figure 5.1 shows a simple example of this using the same scenario as our first Classic Paxos example (Figure 2.2). The proposer p_2 proceeds to phase two of epoch 1 after receiving a promise from one acceptor a_3 since this promise included the proposal $(0, A)$ from the predecessor epoch.

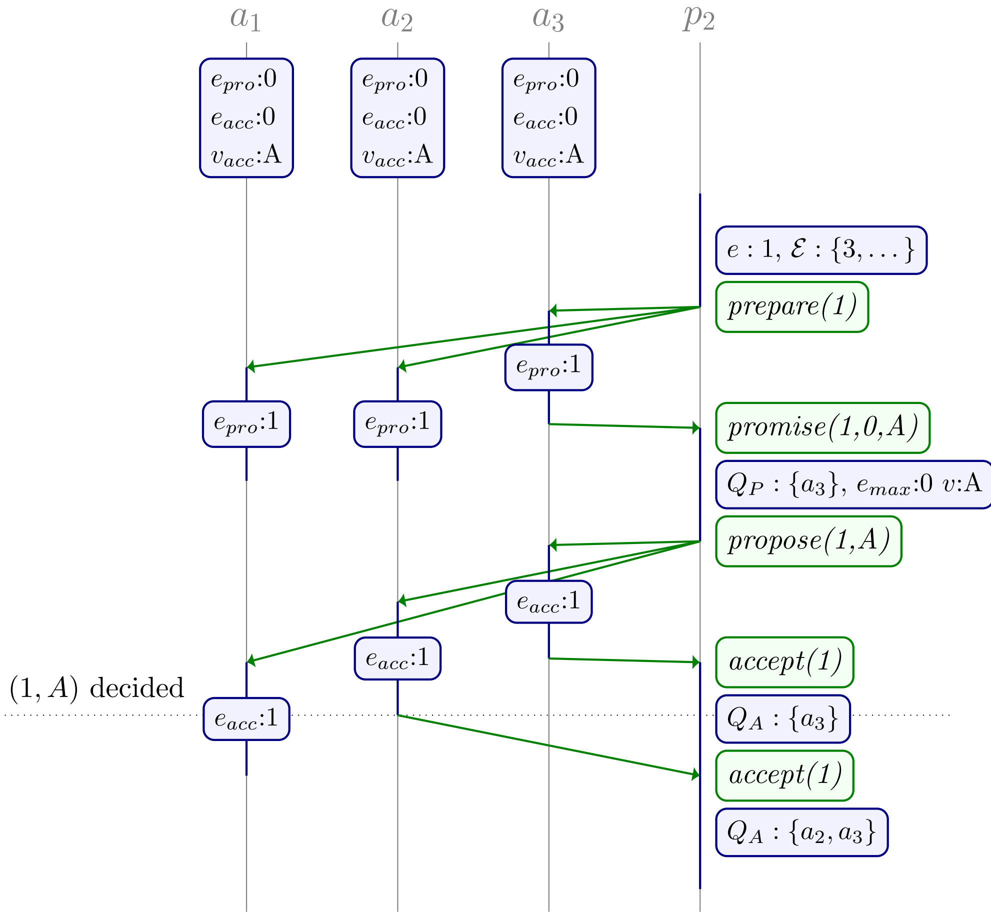

Figure 5.1: Example of a proposer completing phase one early after learning the previous proposal.

Cont. example: Colocating proposers and acceptors

Recall our example of a system of 5 participants, each of which is both an acceptor and proposer (§4.3.2). Paxos revision B allowed us to use the set of acceptors associated only with all previous epochs as a phase one quorum. This is useful in reducing the size of phase one quorums for the first few epochs but quickly becomes useless as the number of previous quorums grows.

We can use Paxos revision C to address this. In our previous systems, consider participant u_4 executing phase one with epoch 3. u_4 can proceed to phase two with less than three promises in the following five scenarios:

- u_4 receives *promise(3,2,_)* from any participant. [1 promise]
- u_4 receives *promise(3,1,_)* from u_3. [1 promise]
- u_4 receives a promise from participant u_3 and *promise(3,1,_)* from any participant. [2 promises]
- u_4 receives a promise from participants u_3 and *promise(3,0,_)* from u_2. [2 promises]
- u_4 receives a promise from participants u_2 and *promise(3,0,_)* from u_3. [2 promises]

5.5 Summary

In this chapter, we prove that proposers may use the transitivity of quorum intersection to re-use the intersection of previous epochs and thus complete phase one prior to satisfying the usual quorum intersection requirement. If a proposer receives a promise with the proposal (e, v) then the proposer no longer needs to intersect with the phase two quorums from epoch up to and including e.

Chapter 6

Value selection revised

In Classic Paxos and our revisions, the value v proposed in phase two is the value associated with the highest epoch, e_{max} received from the acceptors. Initially, e_{max} and v were set to nil and they were updated each time a promise was received which included a proposal with a higher epoch. Once phase one was completed, v was proposed provided it was not nil, otherwise, the proposer's candidate value was proposed. For now on, we refer to this approach as *classic value selection*.

In this section, however, we generalise over the classic value selection rules, by exploiting the additional insight that a proposer gains from each promise it receives. We refer to our revised technique as *Quorum-based value selection* and it can give proposers more flexibility when choosing a value to propose. We divide our discussion into two sections, firstly we consider the simpler case of epoch agnostic quorums (§6.1) before generalising to epoch dependent quorums (§6.1).

6.1 Epoch agnostic algorithm

Algorithm 16 shows an alternative version of Paxos revision A proposer algorithm (Algorithm 13). The acceptor algorithm (Algorithm 4) is unchanged.

Unlike the original algorithm, our new algorithm tracks the promises received from each acceptor in response to *prepare(e)* using R. R is a mapping from each acceptor $a \in A$ to either *no*, meaning that no promise has yet been received or to a proposal (f, w), meaning that *promise(e,f,w)* has been received. Note that as per usual, (f, w) may be nil. Initially, R is set to *no* for all acceptors (line 5, Algorithm 16) and is updated each time a promise is received (line 10, Algorithm 16). Phase one is completed when the proposer has received a promise from at least one acceptor in each phase two quorum (line 7, Algorithm 16).

After which *possibleValues* is passed the set of promises, R, and it returns the set of values

Algorithm 16: Proposer algorithm for Revision A using *possibleValues*.

```
state:
  • R: for each acceptor a ∈ A, either:
      – no: no promise received yet from a
      – (e, v): the proposal received with a promise from a, maybe nil
  • V_dec: set of values which may have been decided

1  v ← nil
2  Q_A ← ∅
3  e ← min(ℰ)
4  ℰ ← ℰ \ {e}
5  ∀a ∈ A : R[a] ← no
   /* Start Phase 1 for epoch e  */
6  send prepare(e) to acceptors
7  while ∃Q ∈ 𝒬_2, ∀a ∈ Q : R[a] = no do
8      switch do
9          case promise(e,f,w) received from acceptor a
10             R[a] ← (f, w)
11         case timeout
12             goto line 1
13 V_dec ← possibleValues(R)
14 if V_dec = ∅ then
15     v ← γ
16 else
17     v ← only(V_dec)
   /* Start Phase 2 for proposal (e,v) */
18 send propose(e,v) to acceptors
19 while ∀Q ∈ 𝒬_2 : Q_A ⊉ Q do
20     switch do
21         case accept(e) received from acceptor a
22             Q_A ← Q_A ∪ {a}
23         case timeout
24             goto line 1
25 return v
```

which may have been decided, V_{dec} (line 13, Algorithm 16)[1]. If V_{dec} is empty, then no decision has been reached and the candidate value is proposed (lines 14-15, Algorithm 16). Otherwise, V_{dec} is a singleton and its only value is proposed (lines 16-17, Algorithm 16). The function *only* returns the only member from a singleton set.

Classic value selection

Algorithm 17: Classic algorithm for *possibleValues*.

1 **func** `possibleValues(`R`):`
2 **return** $\{v \in V | \exists f \in E : R[_] = (f, v)$
3 $\wedge(\forall a \in A : R[a] = no \vee \exists g \in E : R[a] = (g, _) \wedge f \geq g)\}$

Algorithm 17 demonstrates the expected implementation of *possibleValues*, which is equivalent to Classic Paxos and our revisions. The algorithm returns either a set containing the value associated with the greatest proposal or an empty set if all proposals were nil[2].

Revised value selection

Algorithm 18 gives the *Quorum-based* implementation of *possibleValues*. This algorithm is divided into two stages: firstly, it determines whether a decision may have been reached by each quorum and stores the result in D (lines 2-9, Algorithm 18). Then it uses D to determine whether an overall decision may have been reached (line 10, Algorithm 18).

This algorithm for calculating quorum decision (lines 2-9, Algorithm 18) is not simply calculating the highest proposal in each quorum. Instead, it utilises the following two results:

Lemma 19. *If an acceptor a sends* promise(f,e,w) *where* $(e, w) = nil$ *then no decision is reached in epochs up to* f *(exclusive) by the quorums containing* a

Lemma 19 is utilised by lines 3-4 (Algorithm 18) where a proposer sets the decision for a quorum to *no* if any of its acceptors returned *nil* promises.

Lemma 20. *If acceptors* a_1 *and* a_2 *send* promise(g,e,w) *and* promise(g,f,x) *(respectively) where* $e < f$ *and* $w \neq x$ *then no decision is reached in epochs up to* g *(exclusive) by the quorums containing* a_1.

[1] It is not necessary at this point to return a set as it will be either empty or a singleton but we will utilise this later

[2] This algorithm cannot return a set of two or more values since the proposer must have received multiple proposals with the same epoch but different values. We have already shown that this is not possible (Corollary 9.1).

Algorithm 18: Quorum-based algorithm for *possibleValues*.

state:

- D: for each quorum, Q, the outcome of previous proposals, either:
  - no: no decisions have been reached in Q
  - v: if decision(s) were reached in Q, value v was decided

```
1  func possibleValues(R):
2     foreach Q ∈ 𝒬₂ do
3        if ∃a ∈ Q : R[a] = nil then
            /* if acceptor in quorum returns nil then no decision */
4           D[Q] ← no
5        else if ∃a ∈ Q, ∃f, g ∈ E, ∃w, x ∈ V :
6        R[a] = (f, w) ∧ R[_] = (g, x) ∧ g > f ∧ x ≠ w then
            /* if two acceptors return proposals with different
               values then no decision for quorums containing the
               acceptor with the lower proposal */
7           D[Q] ← no
8        else
            /* all proposals returned by quorum are for the same
               value thus this value maybe decided */
9           D[Q] ← only({w ∈ V | ∃a ∈ Q : R[a] = (_, w)})
10    return {w ∈ V | ∃Q ∈ 𝒬₂ : D[Q] = w}
```

Lemma 20 is utilised by lines 5-7 (Algorithm 18) where a proposer sets the decision for a quorum to *no* if any acceptor returned a proposal with a greater epoch and different value to one returned by an acceptor within the quorum.

For a given quorum, Q, if neither of the previous cases (lines 3-7, Algorithm 18) are satisfied then a decision may have been reached in Q. When this case is reached (lines 8-9), then exactly one value has been returned with all the promises from acceptors in Q. We know this because at least one acceptor in Q has promised[3], all acceptors in Q which promised returned a non-nil proposal and if two acceptors returned different values then this case would not be reached.

If a value has been decided, then both implementations of *possibleValues* will return the decided value. If no value has been proposed, both approaches will return an empty set. If exactly one acceptor from each quorum promises then both approaches return same results.

However, if more promises are received, the classic implementation of *possibleValues* may return a value, where the quorum-based implementation may return an empty set. In other

[3]Since possibleValues is only called after at least one acceptor from each quorum has replied.

words, the classic approach may propose a value that the quorum-based approach knows to be undecided[4]. In this implementation, the proposer always proposes its candidate value if no decisions had been reached. However, the proposer could safely propose any value it has seen thus quorum-based value selection is a generalisation over the classic value selection rules.

6.1.1 Safety

We will begin by proving the safety of our epoch agnostic, quorum-based value selection algorithm. Recall that all our earlier proofs of safety depend upon Property 4:

Property 4. *Proposers must choose a value to propose according to the value selection rules. If no previously accepted proposals were returned with promises then any value can be chosen. If one or more previously accepted proposals were returned then the value associated with the highest epoch is chosen.*

This property is implemented by our naïve implementation of possibleValues (Algorithm 17) but not by our quorum-based implementation (Algorithm 18). For quorum-based Paxos, we revise the value selection rule as follows. All other Paxos revision A properties still hold.

Property 16. *Proposers must choose a value to propose in epoch e according to the value selection rules. If V_{dec} is an empty set then any value can be chosen. Otherwise if V_{dec} is a singleton then its only value is chosen.*

We begin be revising our proof of Corollary 12.1.

Corollary 12.1 (Base case for safety of future proposals)**.** *If the value v is decided in epoch e and the value w is proposed $succ(e)$ then $v = w$.*

Revised proof of Corollary 12.1. Assume that (e, v) has been decided and $(succ(e), w)$ has been proposed.

Since (e, v) has been decided, there exists a quorum $Q \in Q_2$ such that all acceptors have accepted (e, v).

The value w which is proposed in $succ(e)$ will have been chosen in one of two ways: either V_{dec} was empty (and w was the proposer's candidate value) or $V_{dec} = \{w\}$ (Property 16). The former case requires that $D[Q] = no$ and the latter requires that either $D[Q] = no$ or $D[Q] = w$ when the proposer of $succ(e)$ finishes phase one. We will now consider each case:

Consider the case that $D[Q] = no$.

[4] The converse is not true.

There are two routes for setting $D[Q] = no$, firstly if any acceptor in Q returns a nil proposal (Algorithm 18, lines 3-4) and secondly if a proposal for a greater epoch with a different value is returned (Algorithm 18, lines 6-7). Since all acceptors in quorum Q have accepted (e, v) prior to promising in $succ(e)$ (Lemma 10) then none will return nil proposals ruling out the former (Lemmas 6 & 7). From lemma 8.1, we know that epoch e is the greatest epoch which will be returned in the proposals thus ruling out the latter. Therefore $D[Q] \neq no$.

Consider the case that $D[Q] = w$.

This case requires that an acceptor in Q has accepted w in some epoch $\leq e$ (Lemma 8.1). Since all accepters in Q have accepted (e, v) then $v = w$ due to value uniqueness (Lemma 9) and monotonicity of accepted epochs (Lemmas 6 & 7). □

Next we revise our proof of Corollary 12.2.

Corollary 12.2 (Inductive case for safety of future proposals)**.** *If the value v is decided in epoch e and the proposals from e (exclusive) to f (inclusive) are limited to the value v then if value w is proposed in g such that $g = succ(f)$ then $v = w$.*

Revised proof of Corollary 12.2. Assume that (e, v) has been decided thus there exists $Q \in Q_2$ such that all acceptors have accepted (e, v). Assume that all proposals in epochs from e to f (inclusive) are for v also.

Assume that $(succ(f), w)$ has been proposed. The value w will have been chosen in one of two ways: either V_{dec} was empty (and w was the proposer's candidate value) or $V_{dec} = \{w\}$ (Property 16). The former case requires that $D[Q] = no$ and the latter requires that either $D[Q] = no$ or $D[Q] = w$ when the proposer of $succ(f)$ finishes phase one. We will now consider each case:

Consider the case that $D[Q] = no$.

There are two possibilities for setting $D[Q] = no$, firstly if any acceptor in Q returns a nil proposal (Algorithm 18, lines 3-4) and secondly if a proposal for a greater epoch with a different value is returned (Algorithm 18, lines 6-7).

All acceptors in quorum Q have accepted (e, v) prior to promising in $succ(f)$ as $succ(f) > e$ (Lemma 10). Therefore none will return nil proposals, thus ruling out the former (Lemmas 6 & 7). From lemma 8.1, we know that epoch f is the greatest epoch which will be returned in the proposals. Likewise, from the monotonicity of accepted proposals (Lemmas 6 & 7), we know that acceptors in Q will return proposals from epochs $\geq e$. Since the epochs e to f are limited to value v then a different value cannot be returned, ruling out the latter. Therefore $D[Q] \neq no$.

Consider the case that $D[Q] = w$.

This case requires that an acceptor in Q has accepted value w in some epoch $\leq f$ (Lemma 8.1). Since all acceptors in Q have accepted (e, v) from the monotonicity of accepted proposals (Lemmas 6 & 7), we know that acceptors in Q will return proposals from epochs $\geq e$. Since the epochs e to f are limited to value v then proposals returned by acceptors in Q must be for value v. Therefore $v = w$. □

In this section, we have proven the safety of our new revision A algorithm using quorum-based value selection (Algorithm 16). We could extend this algorithm to utilise the results of revisions B and C by bypassing phase one when $e = min(E)$ and finishing phase one if a proposal with the predecessor of e is received.

Next, we will prove the correctness of the two results (Lemmas 19 & 20) utilised by Quorum-based value selection (§3).

Lemma 19. *If an acceptor a sends* promise(f,e,w) *where $(e, w) = nil$ then no decision is reached in epochs up to f (exclusive) by the quorums containing a*

Proof of Lemma 19. Assume that an acceptor a sends *promise(f,e,w)* where $(e, w) = nil$.

Prior to sending *promise(f,e,w)*, the acceptor a cannot have accepted any proposal for epochs up to f since $(e, w) = nil$. As such no quorum containing a can have decided a proposal with an epoch up to f.

Subsequently to sending *promise(f,e,w)*, the acceptor a will not have accepted any proposal for epoch up to f as its last promised epoch will always be f or greater. As such no quorum containing a will have decided a proposal with epoch up to f. □

Recall Theorem 13:

Theorem 13 (Safety of future proposals). *If the value v is decided in epoch e and the value w is proposed in epoch f such that $e < f$ then $v = w$*

For a proposal to be accepted it must have been proposed, therefore it follows that:

Corollary 20.1. *If an acceptor a sends* promise(f,e,w) *where $(e, w) \neq nil$ then if a decision is reached in epochs $\leq e$ then value w is chosen.*

Lemma 21. *If two acceptors a_1 and a_2 send* promise(g,e,w) *and* promise(g,f,x) *(respectively) where $e < f$ and $w \neq x$ then no decision is reached in epochs up to e (inclusive).*

Proof of Lemma 21. Assume that acceptor a_1 replies to *prepare(g)* with *promise(g,e,w)* where $(e, w) \neq nil$. Likewise, assume that acceptor a_2 replies to *prepare(g)* with *promise(g,f,x)* where $(f, x) \neq nil$. Assume that $e < f$ and $w \neq x$.

From Corollary 20.1, if a value v is decided with epoch $\leq e$ then $v = w$. Likewise, if a value v is decided with epoch $\leq f$ then $v = x$.

Since $e < f$, then if a value v is decided with epoch $\leq e$ then $v = w$ and $v = x$. This is only satisfied if $w = x$. Hence we have a contradiction. □

Lemma 20. *If acceptors a_1 and a_2 send* promise(g,e,w) *and* promise(g,f,x) *(respectively) where $e < f$ and $w \neq x$ then no decision is reached in epochs up to g (exclusive) by the quorums containing a_1.*

Proof of Lemma 20. Assume that acceptor a_1 sends *promise(g,e,w)* and acceptor a_2 sends *promise(g,f,x)*. Show that a decision (or decisions) could be reached in epoch up to g (exclusive) by a quorum Q containing a_1.

From Lemma 21, we know that no decision could be reached in epoch $\leq e$. Since acceptor a_1 has sent *promise(g,e,w)* then it cannot accept proposals from e to g (exclusive) thus no decision can be reached by quorum Q as $a_1 \in Q$. □

6.1.2 Progress

Our epoch agnostic, quorum-based value selection algorithm relies on the fact that a set is a singleton, each time it uses the *only* function. This occurs in two places: line 17 of Algorithm 16 and line 9 of Algorithm 18. If this is not the case, the proposer algorithm will halt, reaching deadlock and violating our progress guarantees. In this section we will prove that this cannot occur.

Lemma 22. *A value is always returned by the assignment on line 9 of Algorithm 18.*

Proof of Lemma 22. We require that the set passed to *only* must be a singleton. We will prove this by contradiction by showing that neither an empty set or a set of cardinality > 1 could be passed to *only* on line 9 of Algorithm 18.

Consider the case that for some quorum Q, $\{w \in V | \exists a \in Q : R[a] = (_, w)\} = \emptyset$.

This requires that for all acceptors in the quorum Q, $R[a] = nil$ or $R[a] = no$. *possibleValues* is only called after $R[a] \neq no$ for least one acceptor from each quorum. The if-statement on line 3 was false, thus for all acceptors in Q, $R[a] \neq nil$. Thus this case cannot occur.

Consider the case that for some quorum Q, $|\{w \in V | \exists a \in Q : R[a] = (_, w)\}| > 1$.

This requires that (at least) two acceptors in the same quorum return proposes for different values. Since the if-statement on line 5 was false, these acceptors must have returned proposals for the same epoch (due to the total ordering of epochs). This case cannot occur due to value uniqueness (Corollary 9.1). □

Lemma 23. *A value is always returned by the assignment on line 17 of Algorithm 16.*

Proof of Lemma 23. We require that V_{dec}, the set passed to *only* on line 17 of Algorithm 16 must be a singleton. Since the if-statement on line 14 was false, $V_{dec} \neq \emptyset$. We must therefore prove that $|\{w \in V | \exists Q \in \mathcal{Q}_2 : D[Q] = w\}| \leq 1$ (line 10, Algorithm 18).

Proof by contradiction. Assume that two (or more) quorums, Q and Q', have different values for $D[Q]$. This requires that two acceptors, one in Q and one in Q', promised with proposals for different values (line 9, Algorithm 18). If the epoch of these proposals are different, then the quorum with lower epoch would have $D[Q] = no$. Therefore, we require that the epochs of these proposals are the same, however, this cannot occur due to value uniqueness (Corollary 9.1). □

6.1.3 Examples

Consider the following example where $A = \{a_1, a_2, a_3, a_4, a_5\}$ and $\mathcal{Q}_2 = \{\{a_1, a_2, a_3\}, \{a_4, a_5\}\}$. A proposer is in epoch 5 and receives the following promises (in order):

- *promise(5,3,A)* from a_1, followed by
- *promise(5,nil,nil)* from a_2, followed by
- *promise(5,2,B)* from a_4

In Classic Paxos, the proposer must propose the value associated with the highest epoch, in this case A. However, utilising quorum-based value selection a proposer can learn that no decision has been reached in the epochs $0-4$ and thus the proposer is free to choose any value for phase two. As a_2 returned a nil proposal, the quorum $\{a_1, a_2, a_3\}$ cannot reach a decision in epochs $0-4$. Likewise, since a_1 returned the proposal $(3, A)$ then the proposer learns that $(2, B)$ cannot have been decided thus the quorum $\{a_4, a_5\}$ also cannot reach a decision in epochs $0-4$.

Note that this generalisation is also useful when no promises include nil proposals. Consider the case when the proposer instead receives the following promises (in order):

- *promise(5,3,A)* from a_1, followed by
- *promise(5,1,A)* from a_2, followed by
- *promise(5,2,B)* from a_4.

As before, the usual Paxos algorithm would require the proposer to propose A, however, with quorum-based value selection, the proposer is free to propose any value. This is because the quorum $\{a_1, a_2, a_3\}$ cannot have reached a decision since the proposal $(2, B)$ means that the proposal $(1, A)$ from a_2 cannot have been decided. Likewise the quorum $\{a_4, a_5\}$ cannot have reached a decision due to the proposal $(3, A)$.

6.2 Epoch dependent algorithm

Thus far we have introduced quorum-based value selection as an alternative to Paxos's classic value selection rule. Our algorithm for this utilises our earlier work on Paxos revision A. However, we were only able to make limited use of revision B and C since the same quorums are used for all epochs. In this section, we see how proposers can track promises not only by quorum but also by epochs. This generalisation allows us to vary quorums depending on the epoch.

Algorithm 19: Proposer algorithm for Revision B/C using *possibleValues*.

```
1  v ← nil
2  Q_A ← ∅
3  e ← min(ℰ)
4  ℰ ← ℰ \ {e}
5  ∀a ∈ A : R[a] ← no
6  V_dec ← possibleValues(R,e)
   /* Start Phase 1 for epoch e */
7  send prepare(e) to acceptors
8  while |V_dec| > 1 do
9      switch do
10         case promise(e,f,w) received from acceptor a
11             R[a] ← (f, w)
12             V_dec ← possibleValues(R,e)
13         case timeout
14             goto line 1
15 if V_dec = ∅ then
16     v ← γ
17 else
18     v ← only(V_dec)
   /* Start Phase 2 for proposal (e,v) */
19 send propose(e,v) to acceptors
20 while ∀Q ∈ 𝒬_2 : Q_A ⊉ Q do
21     switch do
22         case accept(e) received from acceptor a
23             Q_A ← Q_A ∪ {a}
24         case timeout
25             goto line 1
26 return v
```

Algorithms 19 and 20 gives an implementation of quorum-based value selection for Paxos

Algorithm 20: Quorum-based algorithm for *possibleValues* (Revision B/C).

state:

- D: for each quorum, Q, in each epoch, e, the outcome of previous proposals, either:
  - *no*: no decision has been reached in Q during e
  - v: if a decision was reached in Q during e, value v was decided
  - *nil*: no information known on whether a decision was reached in Q during e

1 **func** `possibleValues(`R,e`):`
2   **foreach** $f \in \{f \in E | f < e\}$ **do**
3   **foreach** $Q \in \mathcal{Q}_2^f$ **do**
4   **if** $\exists a \in Q : R[a] = nil$ **then**
        `/* if an acceptor in the quorum returns nil then no decision */`
5         $D[Q] \leftarrow no$
6   **else if** $\exists a \in Q, \exists g \in E : g < f \wedge R[a] = (g, _)$ **then**
        `/* if an acceptor in the quorum returns lower proposal then no decision */`
7         $D[Q] \leftarrow no$
8   **else if** $\exists g, h \in E, \exists w, x \in V : R[_] = (g, w) \wedge R[_] = (h, x) \wedge f \leq g \wedge f \leq h \wedge w \neq x$ **then**
        `/* if two (or more) different proposals returned with` $\geq f$ `then no decisions */`
9         $D[Q] \leftarrow no$
10   **else if** $\exists g \in E, \exists w \in V : R[_] = (g, w) \wedge f \leq g$ **then**
        `/* if one (or more) same proposals returned with` $\geq f$ `then quorum may decide its value */`
11        $D[Q] \leftarrow w$
12   **else**
13        $D[Q] \leftarrow nil$
14 **if** $\exists f \in E, \exists Q \in \mathcal{Q}_2^f : f < e \wedge D[Q] = nil$ **then**
15   **return** V
16 **else**
17   **return** $\{v \in V | \exists f \in E, \exists Q \in \mathcal{Q}_2^f : f < e \wedge D[Q] = v\}$

revisions B and C.

6.2.1 Safety

In this section, we will prove the safety of our epoch dependent, quorum-based value selection algorithm. As with our safety proof for the epoch agnostic algorithm (§6.1.1), we will substitute Property 4 with Property 16.

Property 4. *Proposers must choose a value to propose according to the value selection rules. If no previously accepted proposals were returned with promises then any value can be chosen. If one or more previously accepted proposals were returned then the value associated with the highest epoch is chosen.*

Property 16. *Proposers must choose a value to propose in epoch e according to the value selection rules. If V_{dec} is an empty set then any value can be chosen. Otherwise if V_{dec} is a singleton then its only value is chosen.*

We begin be revising our proof of Corollary 12.1.

Corollary 12.1 (Base case for safety of future proposals)**.** *If the value v is decided in epoch e and the value w is proposed $succ(e)$ then $v = w$.*

Revised proof of Corollary 12.1. Assume that (e, v) has been decided and $(succ(e), w)$ has been proposed.

Since (e, v) has been decided there exists a quorum $Q \in Q_2^e$ such that all acceptors have accepted (e, v).

The value w which is proposed in $succ(e)$ will have been chosen in one of two ways: either V_{dec} was empty (and w was the proposer's candidate value) or $V_{dec} = \{w\}$ (Property 16).

Consider the case that $V_{dec} = \emptyset$.

This requires that for all quorums, for epochs less than $succ(e)$, $D[Q] = no$, including the quorum Q which accepted (e, v). Due to message ordering (Lemma 10) and the monotonicity of promises (Lemmas 6 & 7), $D[Q]$ will not be assigned by either lines 4-5 or 6-7. Due to value uniqueness (Lemma 9) and promise format (Lemma 8.1), $D[Q]$ will not be assigned by lines 8-9. Therefore, $V_{dec} = \emptyset$ cannot occur.

Consider the case that $V_{dec} = \{w\}$.

This requires that for all quorums, for epochs less than $succ(e)$, either $D[Q] = w$ or $D[Q] = no$. We have already shown that for the quorum which accepted (e, v), $D[Q] \neq no$ thus $D[Q] = w$. Since e is the greatest epoch which will be returned with a promise (Lemma 8.1) then $w = v$. □

Next we revise our proof of Corollary 12.2.

Corollary 12.2 (Inductive case for safety of future proposals). *If the value v is decided in epoch e and the proposals from e (exclusive) to f (inclusive) are limited to the value v then if value w is proposed in g such that $g = succ(f)$ then $v = w$.*

Revised proof of Corollary 12.2. Assume that (e, v) has been decided thus there exists $Q \in Q_2$ such that all acceptors have accepted (e, v). Assume that all proposals in epochs from e to f (inclusive) are for v also.

The value w which is proposed in $succ(f)$ will have been chosen in one of two ways: either V_{dec} was empty (and w was the proposer's candidate value) or $V_{dec} = \{w\}$ (Property 16).

Consider the case that $V_{dec} = \emptyset$.

This requires that for all quorums, for epochs less than $succ(e)$, $D[Q] = no$, including the quorum Q which accepted (e, v). Due to message ordering (Lemma 10) and the monotonicity of promises (Lemmas 6 & 7), $D[Q]$ will not be assigned to *no* by either lines 4-5 or 6-7.

Since f is the greatest epoch which will be returned with a promise (Lemma 8.1) and all proposals for epochs e to f are for value v, $D[Q]$ will not be assigned by lines 8-9. Therefore, the case that $V_{dec} = \emptyset$ cannot occur.

Consider the case that $V_{dec} = \{w\}$.

This requires that for all quorums, for epochs less than $succ(e)$, either $D[Q] = w$ or $D[Q] = no$. We have already shown that for the quorum Q which accepted (e, v), $D[Q] \neq no$ thus $D[Q] = w$. As before, f is the greatest epoch which will be returned with a promise (Lemma 8.1). Therefore at least one acceptor in Q will have promised with the proposal (h, w) for some h where $e \leq h \leq f$ and some value w. As all proposals for epochs e to f are for the value v then it must be case that $v = w$. □

6.2.2 Progress

Unlike our epoch agnostic algorithm, in our new algorithm (Algorithm 19) the proposer re-calculates V_{dec} after each promise is received. The proposer then uses the cardinality of V_{dec} to determine when phase one is completed. In contrast to the algorithms thus far, it is not clear that the algorithm will always make progress under the expected liveness conditions. In this section, we therefore prove that the proposer's phase one will terminate once the quorum intersection requirements have been satisfied.

Lemma 24. *If a proposer in epoch e has received sufficient promises to satisfy revision C quorum intersection, then for all quorums of previous epochs $D[Q] \neq nil$.*

Proof of lemma 24. Consider any epoch f where $f < e$ and any one of its phase two quorums Q where $Q \in \mathcal{Q}_2^f$. Prove that $D[Q] \neq nil$. There are two mechanisms by which the revision C quorum intersection requirement with Q may be satisfied.

Consider the case that an acceptor has promised in e with the proposal (g, w) for some epoch g where $g \geq f$ and for some value w.

$D[Q]$ will be set to either *no* or w, depending on whether additional proposals for epochs $\geq f$ and with different value have been received (Algorithm 20, lines 8-11).

Consider the case that an acceptor $a \in Q$ has promised in e with the proposal (g, w) for some epoch g and some value w.

Consider the case that $(g, w) = nil$.

$D[Q]$ will be set to *no* (Algorithm 20, lines 4-5).

Consider the case that $(g, w) \neq nil$.

Due to the total ordering of epochs, either $g < f$ or $g \geq f$. If $g < f$ then $D[Q]$ will be set to *no* (Algorithm 20, lines 6-7). Otherwise $g \geq f$ and we have another instance of our first case. □

Lemma 25. *If a proposer in epoch e has received sufficient promises to satisfy revision C quorum intersection, then V_{dec} is either an empty set or a singleton set.*

Proof of lemma 25. V_{dec} is set to the output of possibleValues (Algorithms 19, line 12). From lemma 24, we know that the if-statement on line 14 of Algorithm 20 will be false. Thus, the output of possibleValues is determined by the return statement on line 17 of Algorithm 20.

Proof by contradiction. Assume that there exist two quorums, from epochs f and g where $D[Q]$ has values, w and x such that $w \neq x$. From the total ordering of epochs, it must either be the case that $f = g$, $f > g$ or $f < g$.

Consider the case that $f = g$.

From value uniqueness (Lemma 9), we know that only one value can be proposed per epoch thus $w = x$.

Consider the case that $f > g$.

For the quorum in epoch g, $D[Q]$ will only be set to value x if there is no proposal with a higher epoch and different value (Algorithm 20, lines 8-9). This cannot be true since we have assumed that $w \neq x$.

The same applies to epoch f if $f < g$. □

6.3 Summary

Classic Paxos (and our revisions) require that, after receiving sufficient promises to satisfy the quorum intersection requirement, the proposer proposes the value associated with the greatest epoch received or its own candidate value if no such values are received.

We proved that by tracking the status of each quorum, proposers can utilise additional promises to remove the requirement that a particular value is proposed in phase two. In this case, the proposer may propose its own candidate value or any previously seen value.

Quorum-based value selection generalises over Classic Paxos's value selection rules. The original rules are a quick and safe approximation of the more complete quorum-based rules. This relationship is analogous to that of Classic Paxos's quorum intersection requirement and Paxos version B requirement.

Chapter 7

Epochs revised

In this chapter, we consider the alternatives to requiring pre-allocated unique epochs, as specified in the earlier description of Classic Paxos (Chapter 2). Thus far, we have relied on the fact that the proposers will not dispatch *propose(e,v)* for the same epoch e and different values v. This could be achieved by *a priori* allocation of epochs between the proposers, thus each proposer may use only a disjoint subset of epochs and requiring each proposer to use each epoch only once. We have also shown that this can be achieved by voting for epochs during phase one of the proposer algorithm (introduced in §3.9, generalised in §4.3.4).

However, the need to allocate epochs to proposers limits what we can achieve with single-valued consensus. In particular, it is desirable for any proposer to be able to decide a value with only one round trip in the best case. Classic Paxos allows any proposer to decide a value in two round trips, though one round trip can be executed prior to knowledge of the value. Paxos revision B enabled the proposer which is able to use the minimum epoch to skip phase one due to the lack of a phase one quorum intersection requirement. However, at most one of the proposers is able to utilise this.

This chapter explores how to overcome this limitation by removing the requirement to pre-allocate or vote for unique epochs, thus providing proposers with more flexibility over the epochs they use. The three approaches discussed are:

- Using an allocator to dynamically allocate epochs (§7.1).
- Pre-allocation of epochs based on the value to be proposed in phase two (§7.2).
- Allowing different values to be proposed with the same epoch but requiring phase two intersection and strengthened intersection requirements across phases (§7.3).

These approaches, in addition to the original techniques (unique epochs by pre-allocation and voting) can be combined on a per-epoch basis to create a hybrid algorithm (§7.4). We will now consider each approach in more detail.

7.1 Epochs from an allocator

Thus far we have assumed that the epochs, E, have been allocated *a priori* between proposers. Instead, we could use an *allocator* to dynamically allocate epochs between proposers. The allocator need be no more complex than a simple counter, starting at e_{min}. We replace the selection of the next epoch (Lines 3-4 in Algorithm 3) with a message exchange with the allocator. The allocator must guarantee that each epoch is allocated at most once[1].

Algorithm 21: Allocator algorithm

state:
- *sid*: sequence number
- *vid*: service version number (persistent, initially 0)

```
1 sid ← 0, vid ← vid + 1
2 while true do
3     switch do
4         case generate-next() received from proposer
5             sid ← sid + 1
6             send allocate((vid,sid)) to proposer
```

Algorithm 21 gives a naïve algorithm to implement the allocator on a single participant. Epochs are an ordered tuples of the form (vid, sid) such as:

$$E = \{(1,1),(1,2),(1,3),\dots,(2,1),(2,2),(2,3),\dots\}$$

The algorithm is effectively a simple counter, *sid*, with a version number, *vid*, to ensure uniqueness of epochs assigned in the case of failure. The service version number, *vid*, is stored in persistent storage and incremented on each restart. The sequence number, *sid*, is stored in volatile storage and incremented on each allocation.[2] Given that an allocator will assign each epoch at most once and will assign epochs in increasing order, our safety proofs still hold without revisions as all previous properties still hold.

We could extend this naïve approach to have proposers include their candidate value in their request to the allocator. The allocator could store the epoch to value mapping. This would enable the allocator to re-allocate epochs to other proposers on the condition that they propose the same value as was originally assigned. This could allow for conflict-free recovery for slow/failed proposers. In this case, e_{min} would be the only epoch allocated by the allocator. The allocator would equivalent to a single write-once register, which is

[1] Note that allocation need not be in-order nor does every epoch need to be allocated for safety. However, similar to Classic Paxos, using epochs in-order does simplify our proof of progress.

[2] This algorithm implements unique epochs without synchronous writes to persistent storage for each proposal, this technique was first described in §3.8.

initially set to nil[3]. This algorithm is the same as the acceptor algorithm for SAA (2.1.1). As before, the safety proofs still hold as all previous properties still hold.

These two simple mechanisms, exclusive epochs from an allocator and a shared epoch from an allocator, allows any proposer to be allocated the minimum epoch, enabling them to bypass phase one. This does, however, require an extra phase to request and receive an epoch. Furthermore, the liveness of the system now also depends on the availability of the allocator, introducing a single point of failure, meaning the algorithms are of little practical use. We will address this limitation later in §7.4.

7.2 Epochs by value mapping

The reason for requiring unique epochs is to ensure proposers cannot dispatch *propose(e,v)* for the same epoch e and different values v. Another mechanism for achieving this is to pre-allocate epochs to their associated value instead of to proposers. A proposer wishing to propose a value v will use its first epoch. If the proposer is unable to choose its own value after executing phase one, it will need to retry phase one with the epoch associated with its expected value.

The advantages of this approach are that proposers do not need to store epochs in persistent storage (as is the case for epochs by pre-allocation) or phase one quorum intersection (as is the case for epochs by voting). The result of this is that any proposer who wishes to propose the value corresponding with the minimum epoch e_{min} may skip phase one.

However, phase one now requires knowledge of the value to be proposed. This also means that the phase one cannot be pre-executed (as described in §3.5). As such more phases may be needed in situations where the proposer changes the value they wish to propose as an outcome of phase one. This approach does not satisfy the problem of distributed consensus as it can only be applied to systems where the values which may be decided are to be within a finite known set, we will address this limitation later in §7.4[4].

Example: Binary consensus algorithm

This approach is best illustrated by considering an algorithm to reach consensus over a binary value, for example on whether a transaction should be *committed* ($v = 1$) or *aborted* ($v = 0$). Algorithms 22 and 23 give example pseudocode for this. We will let odd epochs correspond to $v = 1$ and even epochs correspond to $v = 0$. Since $e_{min} = 0$, we can utilise skipping phase one (from Paxos revision B) so an abort decision could be achieved in

[3] This statement assumes that the value is stored in persistent storage. Otherwise, the allocator would need to allocate new epochs on recovery.

[4] Known infinite value sets can also be supported provided the epoch set is divisible into an infinite number of infinite subsets.

Algorithm 22: Proposer algorithm for binary decision

state:

- γ: candidate value, 1 or 0
- e: current epoch (initially nil)

```
1  e_max ← nil
2  Q_P, Q_A ← ∅
3  if e = nil then
4      if γ = 0 then
5          e ← 0, goto line 25
6      else
7          e ← 1
8  else
9      e ← e + 1
10     if e  mod 2 ≠ γ then
11         e ← e + 1
   /* Start of Phase 1 for proposal e  */
12 send prepare(e) to acceptors
13 while ∃Q ∈ Q_2 : Q_P ∩ Q = ∅ ∧ e_max ≠ e − 1 do
14     switch do
15         case promise(e,f) received from a
16             Q_P ← Q_P ∪ {a}
17             if f ≠ nil ∧ (e_max = nil ∨ f > e_max) then
18                 e_max ← f
19         case timeout
20             goto line 1
21 if e_max ≠ nil ∧ (e_max  mod 2 ≠ e  mod 2) ∨ e_max = nil ∧ (e  mod 2 = γ) then
       /* proposal value does not match epoch so try again */
22     e_max ← nil
23     Q_P ← ∅
24     e ← e + 1, goto line 12
   /* Start of Phase 2 for proposal (e) */
25 send propose(e) to acceptors
26 while ∀Q ∈ Q_2 : Q_A ⊉ Q do
27     switch do
28         case accept(e) received from a
29             Q_A ← Q_A ∪ {a}
30         case timeout
31             goto line 1
32 return e  mod 2
```

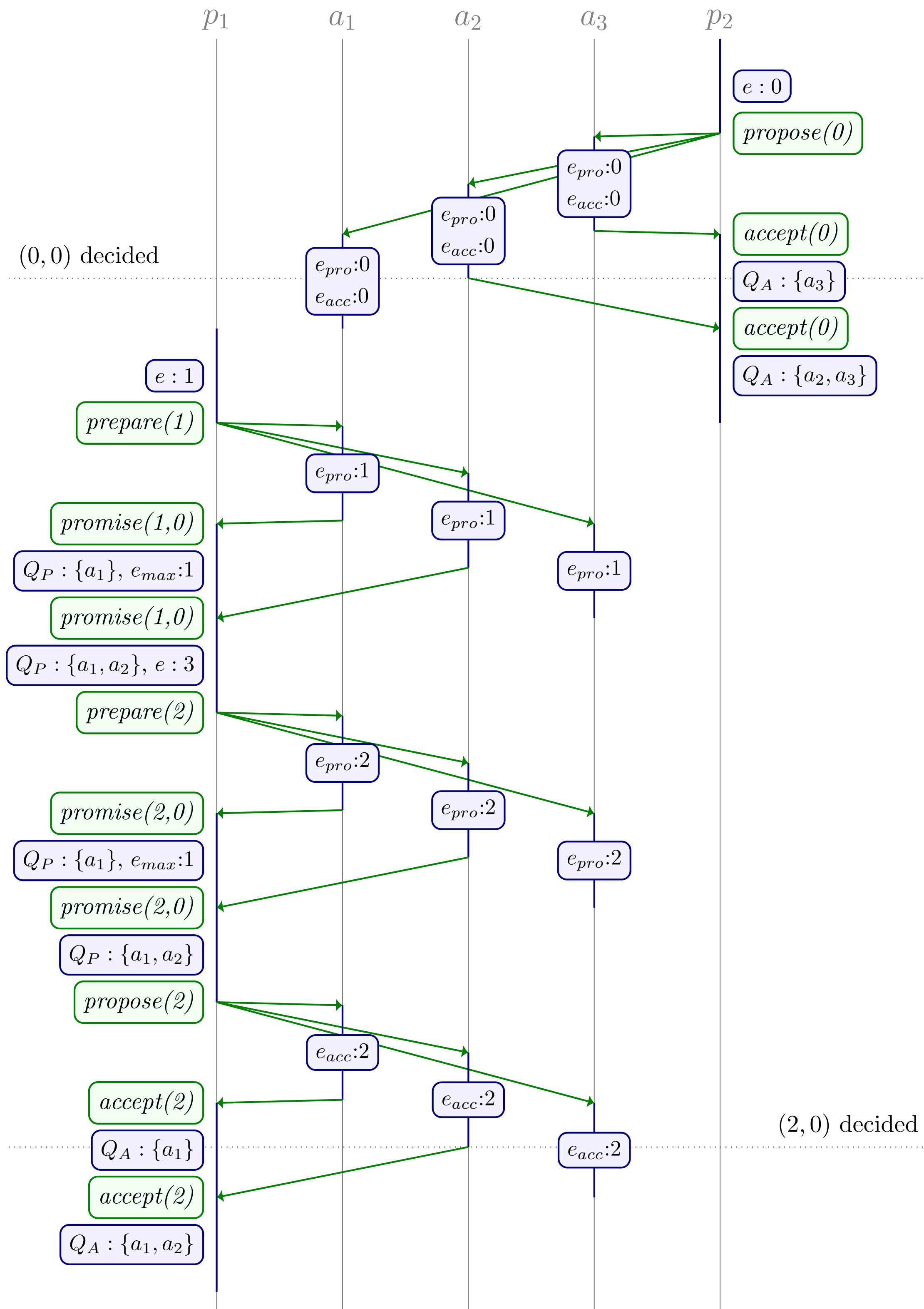

Figure 7.1: Paxos for binary consensus (Alg. 23,22)

Algorithm 23: Acceptor algorithm for binary decision

```
1  while true do
2      switch do
3          case prepare(e) received from proposer
4              if e_pro = nil ∨ e ≥ e_pro then
5                  e_pro ← e
6                  send promise(e,e_acc) to proposer
7          case propose(e) received from proposer
8              if e_pro = nil ∨ e ≥ e_pro then
9                  e_pro ← e
10                 e_acc ← e
11                 send accept(e) to proposer
```

only one phase (phase one) by any proposer (Algorithm 22, lines 4-5). Likewise, we are able to skip the remainder of phase one if a promise is received with a proposal from the predecessor epoch (Algorithm 22, line 13). As the epochs directly correspond to values, $v = e \mod 2$, the proposed/accepted value can also be omitted. Figure 7.1 shows an example of this in practice where proposer p_1 wishes to commit and p_2 wishes to abort.

7.3 Epochs by recovery

In this chapter so far, we have proposed various techniques to maintain value uniqueness (lemma 9). In this section, however, we will consider how to remove the requirement that values are unique to epochs. Our approach which we will refer to as *epochs by recovery*, allows proposers to use any epoch by adding mechanisms to recover if multiple values are proposed for the same epoch.

7.3.1 Intuition

We will now derive an algorithm with shared epochs by considering what will go wrong if we were to simply permit epochs to be shared in Classic Paxos. To maintain generality, we use Paxos revision B as our starting point[5].

Problem 1: Firstly, it is possible that multiple values are committed by different proposers with the same epoch, since each value can be accepted by non-intersecting phase two quorums.

[5] Later, we will consider whether we can also apply revision C as it directly uses the value uniqueness lemma.

Solution 1: Therefore, we require that the phase two quorums of a given epoch intersect, stated as:

$$\forall Q, Q' \in \mathcal{Q}_2^e : Q \cap Q' \neq \emptyset \tag{7.1}$$

Problem 2: Secondly, a value which has already been accepted by a phase two quorum can be overwritten by different values with the same epoch, violating protocol safety.

Solution 2: This can be addressed by adding a condition to phase two that a proposal (e, v) is only accepted if either the new proposal epoch is higher than the previous one $e > e_{acc}$ or the new proposal is the same as the previous one $(e, v) = (e_{acc}, v_{acc})$. In other words, an acceptor cannot overwrite an accepted value with the same epoch.

Problem 3: Thirdly, the approach described thus far may reach a state from which it is unable to make progress under the usual liveness conditions. We refer to this as *value collision*.

Recall that the value selection rules of Paxos require that a proposer chooses the value associated with the highest epoch received in phase one. In the example, the proposer has received two promises in phase one of the algorithm, with the same epoch but two different values. The proposer must choose which of the two values to propose in its phase two. When choosing a value, a proposer must know for certain that no other value has been decided. In this case, however, the proposer cannot know which order the prepare messages were received by other acceptors, if they have been received at all. Therefore, since the proposer cannot safely proceed through the algorithm, it cannot make progress.

Solution 3: This example demonstrates the case for strengthening the quorum intersection requirements when using shared epochs. The previous quorum intersection requirement 4.6 is not necessarily sufficient to make progress as we have seen. The following intersection rule, given in 7.2 is sufficient to always make progress. In Paxos revisions B, we required that a phase one quorum intersects with any previous phase two quorums.

Now, we require that a phase one quorum intersects with the intersection of any phase two quorums for a previous epoch. More formally, for each epoch e the following intersection requirement is sufficient:

$$\forall Q \in \mathcal{Q}_1^e, \forall f \in E : f < e \implies \forall Q', Q'' \in \mathcal{Q}_2^f : Q \cap Q' \cap Q'' \neq \emptyset \tag{7.2}$$

It is worthwhile noting that this quorum intersection rule is an upper bound on the phase one quorum needed in the worst case scenario. The usual weaker requirement 4.6 may be sufficient, depending upon the promises received. As with 4.6, the result of this requirement is that for epoch e_{min}, the minimum epoch, there is no phase one quorum intersection requirement. The result is that any proposer may skip phase one for e_{min}.

7.3.2 Algorithm

Algorithm 24: Acceptor algorithm for epochs by recovery.

```
1  while true do
2      switch do
3          case prepare(e) received from proposer
4              if e_pro = nil ∨ e ≥ e_pro then
5                  e_pro ← e
6                  send promise(e,e_acc,v_acc) to proposer
7          case propose(e,v) received from proposer
8              if e_pro = nil ∨ e ≥ e_pro ∧(e ≠ e_acc ∨ v = v_acc) then
9                  e_pro ← e
10                 v_acc ← v, e_acc ← e
11                 send accept(e,v) to proposer
```

Algorithm 24 outlines an acceptor algorithm for epochs by recovery. The only two differences between this and the Classic Paxos acceptor algorithm are that accept messages now include the value (line 11) and an extra condition on accepting proposals (line 8). Specifically, acceptors do not overwrite accepted proposals with proposals of the same epoch but a different value. This is implemented on line 8 whereupon receiving a propose, the acceptor must check that they have not already accepted a proposal with this number but a different value.

Algorithm 25 outlines a phase one for the proposer algorithm for Revision A with epochs allocated by recovery[6]. We have switched to epoch agnostic, quorum-based value selection (§6.1) as this approach is better suited to efficiently expressing epochs by recovery.

There are three key differences between this algorithm and Revision A with quorum-based value selection (Algorithm 16)[7].

Firstly, as proposers are no longer required to choose from disjoint sets of epochs and track which have been used, $\mathcal{E}$ has been removed. Instead, epoch e is initially set to *nil* and is incremented before each use (line 4-7, Algorithm 25)[8][9].

Secondly, our implementation of *possibleValues* (Algorithm 26) includes an extra case to

[6]For simplicity, we are not varying quorums depending upon epoch thus Revisions B and C do not apply.

[7]Whilst not explicitly represented in the pseudocode, this algorithm also requires that phase two quorums of a given epoch must intersect.

[8]In contrast to our previous algorithms which were general to any epoch set E, it is now the case that $E = \mathbb{N}^0$. This approach has been chosen for simplicity however the algorithms could easily be generalised to any epoch set E.

[9]The current epoch e does not need to be in persistent storage for correctness, however, it would help proposers recovery quickly after failure.

Algorithm 25: Proposer algorithm for Revision A with epochs by recovery.

state:

- e: current epoch (persistent, initially nil)

1 $v \leftarrow nil$
2 $Q_A \leftarrow \emptyset$
3 $V_{dec} \leftarrow \emptyset$
4 **if** $e = nil$ **then**
5 $\quad e \leftarrow 0$, $v \leftarrow \gamma$, **goto** *line 21*
6 **else**
7 $\quad e \leftarrow e + 1$
8 $\forall a \in A : R[a] \leftarrow no$
/* Start Phase 1 for epoch e */
9 **send** *prepare(e) to acceptors*
10 **while** $(\exists Q \in \mathcal{Q}_2, \forall a \in Q : R[a] = no) \vee |V_{dec}| > 1$ **do**
11 $\quad$ **switch do**
12 $\quad\quad$ **case** *promise(e,f,w) received from acceptor a*
13 $\quad\quad\quad R[a] \leftarrow (f, w)$
14 $\quad\quad\quad V_{dec} \leftarrow$ `possibleValues(`R`)`
15 $\quad\quad$ **case** *timeout*
16 $\quad\quad\quad$ **goto** *line 1*
17 **if** $V_{dec} = \emptyset$ **then**
18 $\quad v \leftarrow \gamma$
19 **else**
20 $\quad v \leftarrow only(V_{dec})$
/* Start Phase 2 for proposal (e,v) */
21 **send** *propose(e,v) to acceptors*
22 **while** $\forall Q \in \mathcal{Q}_2 : Q_A \not\supseteq Q$ **do**
23 $\quad$ **switch do**
24 $\quad\quad$ **case** *accept(e ,v) received from acceptor a*
25 $\quad\quad\quad Q_A \leftarrow Q_A \cup \{a\}$
26 $\quad\quad$ **case** *timeout*
27 $\quad\quad\quad$ **goto** *line 1*
28 **return** v

Algorithm 26: Algorithm for *possibleValues* with epochs by recovery (Revision A).

```
1  func possibleValues(R):
2      foreach Q ∈ 𝒬₂ do
3          if ∃a ∈ Q : R[a] = nil then
               /* if acceptor in quorum returns nil then no decision */
4              D[Q] ← no
5          else if ∃a ∈ Q, ∃f, g ∈ E, ∃w, x ∈ V :
6          R[a] = (f, w) ∧ R[_] = (g, x) ∧ g > f ∧ x ≠ w then
               /* if two acceptors return proposals with different
                  values then no decision for quorums containing the
                  acceptor with the lower proposal */
7              D[Q] ← no
8          else if ∃a, b ∈ Q : ∃f ∈ E, ∃w, x ∈ V :
9          R[a] = (f, w) ∧ R[b] = (f, x) ∧ w ≠ x) then
               /* if two acceptors in the same quorum return proposals
                  with same number but different values then no
                  decision */
10             D[Q] ← no
11         else
               /* all proposals returned by quorum are for the same
                  value thus this value maybe decided */
12             D[Q] ← only({w ∈ V | ∃a ∈ Q : R[a] = (_, w)})
13     return {w ∈ V | ∃Q ∈ 𝒬₂ : D[Q] = w}
```

set D for quorum Q to no if two acceptors within the quorum have returned promises with the same epoch but different values (line 7-8, Algorithm 26).

Thirdly, after satisfying the usual quorum intersection requirement, if there are multiple possibly decided values then the proposer must wait for additional promises to rule out values until only one or zero values remain. This is implemented by adding the condition on the cardinality of V_{dec} (line 10, Algorithm 25).

7.3.3 Safety

We will prove the safety of Paxos revisions A with epochs allocated by recovery using the usual method. Our usual properties still hold, except from properties 1 & 4, restated below:

Property 1. *Proposers use unique epochs for each proposal.*

Property 4. *Proposers must choose a value to propose according to the value selection rules. If no previously accepted proposals were returned with promises then any value can be chosen. If one or more previously accepted proposals were returned then the value associated with the highest epoch is chosen.*

However, we will add the following three additional properties for future use:

Property 17. *For each* propose *message received by an acceptor where the epoch received is the same as the last accepted epoch, the message is processed by the acceptor only if the proposed value is the same as the last accepted value.*

Property 18. *Proposers only propose a value after receiving promises from sufficient acceptors such that at most one value may have been decided.*

Property 19. *Proposers must choose a value to propose in epoch e according to the value selection rules. If V_{dec} is an empty set then any value can be chosen. Otherwise if V_{dec} is a singleton then its only value is chosen.*

From property 17 it follows that:

Lemma 26. *An acceptor will not accept more than one proposal with a given epoch. If an acceptor accepts (e, v) and (e, w) for any epoch $e \in E$ then $v = w$.*

Proof of Lemma 26. Assume that an acceptor has accepted (e, v) then (e, w). From Properties 10, 6 & 9, the last accepted proposal must be (e, v) when (e, w) is accepted. From Property 17, then $v = w$. □

We can therefore show that:

Lemma 27. *If the value v is decided in epoch e then no other value w where $v \neq w$ will also be decided in e.*

Proof of lemma 27. Assume the proposal (e, v) has been decided therefore a phase two of acceptors $Q \in \mathcal{Q}_2$ have accepted (e, v). Likewise for w to be decided, a phase two quorum of acceptors $Q' \in \mathcal{Q}_2$ must have accepted (e, w). As any two phase two quorums for a given epoch intersect, then at least one acceptor must have accepted both proposals. From lemma 26 then $v = w$, so no other value can be accepted. □

We begin be revising our proof of Corollary 12.1.

Corollary 12.1 (Base case for safety of future proposals)**.** *If the value v is decided in epoch e and the value w is proposed $succ(e)$ then $v = w$.*

Revised proof of Corollary 12.1. Assume that (e, v) has been decided and $(succ(e), w)$ has been proposed.

Since (e, v) has been decided thus there exists a quorum $Q \in Q_2$ such that all acceptors have accepted (e, v).

The value w which is proposed in $succ(e)$ will have been chosen in one of two ways: either V_{dec} was empty (and w was the proposer's candidate value) or $V_{dec} = \{w\}$ (Property 19). The former case requires that $D[Q] = no$ and the later requires that either $D[Q] = no$ or $D[Q] = w$ when the proposer of $succ(e)$ finishes phase one. We will now consider each case

Consider the case that $D[Q] = no$.

Since all acceptors in quorum Q have accepted (e, v) then none will return nil proposals with promises (lines 3/4). Likewise, acceptors will not accept another proposal from e (lines 9/10). Thus another acceptor must have returned a proposal for an epoch $> e$ (lines 8-10, Property 19). This epoch must be $succ(e)$.

Consider the case that $D[Q] = w$.

Since all acceptors in quorum Q have accepted (e, v) either $w = v$ or $w = x$ where $(succ(e), x)$ has been proposed.

We have seen that either $w = v$ or $w = x$ where x is another value which has been proposed in epoch $succ(e)$. If this is first value proposed in $succ(e)$ then it must be the case that $w = v$. If all other values proposed in $succ(e)$ are v then $w = v$. They we have proven 12.1 by induction. □

Next we revise our proof of Corollary 12.2.

Corollary 12.2 (Inductive case for safety of future proposals)**.** *If the value v is decided in epoch e and the proposals from e (exclusive) to f (inclusive) are limited to the value v then if value w is proposed in g such that $g = succ(f)$ then $v = w$.*

Revised proof of Corollary 12.2. Assume that (e, v) has been decided thus there exists $Q \in Q_2$ such that all acceptors have accepted (e, v). Assume that all values proposed in epochs from e to f are for v also.

Assume that value w has been proposed by a proposer in epoch $succ(f)$. The value w will have been chosen in one of two ways: either V_{dec} was empty (and w was the proposer's candidate value) or $V_{dec} = \{w\}$.

Consider the case that $V_{dec} = \emptyset$.

For all quorums including Q, $D[Q] = no$. Given that all acceptors in Q have accepted (e, v), it is only possible for $D[Q] = no$ if an acceptor returns *promise(succ(f),h,x)* where $h > e$ and $x \neq v$ (Property 19). As all values proposed in epochs from e to f are for v then $h = succ(f)$. By induction, we can see that $x = v$ thus this case cannot occur.

Consider the case that $V_{dec} = \{w\}$.

For all quorums including Q, $D[Q] = no$ or $D[Q] = w$. As we have already shown it cannot be the case that $D[Q] = no$ thus $D[Q] = w$ and therefore for some acceptor $\exists a \in Q : R[a] = (h, w)$. This acceptor must have first accepted (e, v) thus $h \geq e$. If $h = e$ then $v = w$ (Lemma 26). Otherwise, if $e < h \leq f$ then $v = w$ as all values proposed in these epochs are v. Otherwise $h = succ(f)$ and by induction we can see that $w = v$. □

As before, theorems 12.1 and 12.2 will form the base and inductive case for proving theorem 13.

Proof of safety of Classic Paxos

Overall, to prove the safety of Paxos, we wish to show that:

Theorem 14 (Safety for Classic Paxos)**.** *If the value v is decided in epoch e and the value w is decided in epoch f then $v = w$*

Revised proof of theorem 14. Consider the case that $e = f$.

Theorem 27 shows that at most once value will be decided in a given epoch.

Consider the case that $e \neq f$.

Since there is a total ordering on epochs then either $e < f$ or $e > f$. From the symmetry of theorem 14, we can assume $e < f$ and derive $e > f$ by swapping e and f.

For a value to be decided, it must first be proposed, therefore a stronger theorem is theorem 13. □

7.3.4 Progress

Earlier, we claimed that the strengthened quorum intersection requirement in Equation 7.2 is always sufficient to make progress. Now we will examine this claim.

Lemma 28. *After a proposer in epoch e has received sufficient promises to satisfy Equation 7.2,* possibleValues *always returns an empty or singleton set.*

Proof of lemma 28. Consider a proposer in epoch e who is calling *possibleValues* after receiving sufficient promises to satisfy Equation 7.2. Assume that *possibleValues* returns a set of two or more values such as $\{v, v', ...\}$ where $v \neq v'$.

It is therefore the case that there are two quorums $Q, Q' \in \mathcal{Q}_2$ such that $D[Q] = v$ and $D[Q'] = v'$.

This requires that $\forall a \in Q : R[a] = no \vee R[a] = (_, v)$ and $\forall a \in Q' : R[a] = no \vee R[a] = (_, v')$.

From equation 7.2, we know that $\exists a \in A : R[a] \neq no \wedge a \in Q \wedge a \in Q'$. Combining this with the above result, we learn that $\exists a \in A : R[a] = (_, v) \wedge R[a] = (_, v')$. This requires that $v = v'$ thus we have a contradiction. □

7.3.5 Examples

We will now examine three examples of epochs by recovery using three different classes of quorums systems.

Example: All aboard Paxos with epochs by recovery

Algorithm 27: Proposer algorithm with epochs by recovery and a fixed quorum.

state:
- Q: fixed phase two quorum

```
1  Q_A ← ∅
2  if e = nil then
3      e ← 0
4  else
5      e ← e + 1
   /* Start of Phase 1 for proposal e */
6  send prepare(e) to acceptors
7  switch do
8      case promise(e,_,w) received from acceptor a ∈ Q
9          if w ≠ nil then
10             v ← w
11         else
12             v ← γ
13     case timeout
14         goto line 1
   /* Start Phase 2 for proposal (e,v) */
15 send propose(e,v) to acceptors
16 while Q_A ⊉ Q do
17     switch do
18         case accept(e ,v) received from acceptor a
19             Q_A ← Q_A ∪ {a}
20         case timeout
21             goto line 1
22 return v
```

Algorithms which use epochs by recovery need not be complex. For example, the simplest quorum system contains a single fixed quorum, Q. If we let $\mathcal{Q}_2 = \{Q\}$ then one promise from an acceptor in Q will always be sufficient to complete phase one, as shown in Algorithm 27. This algorithm may be simple but it does require all acceptors in Q to be up for liveness. This algorithm is similar to the first iteration of All aboard Paxos (§4.3.2) with the added flexibility that acceptors may use any epoch.

Example: Fixed quorums for epochs by recovery

Instead, Algorithm 28 assigns a single quorum, Q^e to each epoch e such that $\forall e \in E$: $\mathcal{Q}_2^e = \{Q^e\}$. All phase two quorums in epoch e are guaranteed to intersect at all acceptors in Q^e, therefore a single promise from an acceptor in Q^e is sufficient to satisfy the strengthened intersection requirement. We have also applied Paxos revision C to this algorithm. Algorithm 27 is a special case of Algorithm 28 where each epoch is assigned the same quorum.

Note that this proposer algorithm is very similar to Paxos revision C proposer algorithm when each epoch has only one phase two quorum. The key difference here is that receiving a proposal (f, v) is only sufficient to satisfy the quorum intersection requirement for epochs strictly less than f, unlike Paxos revision C where this was sufficient for epoch less than or equivalent to f. Aside from this, shared epochs are effectively free as no additional promises are needed.

Example: Counting quorums for epochs by recovery

Our algorithm for epochs by recovery (Algorithm 25) was quorum system agnostic. In this section, we specialise the algorithm for *counting quorums*, where any set of k or more acceptors is a phase two quorum. A pseudocode proposer algorithm is shown in Algorithm 29. The acceptor remains unchanged. Since we require phase two quorum intersection (Equation 7.1) then we require that $2k > n_a$ where n_a is the number of acceptors and k is the quorum size.

There are two conditions which must be satisfied to complete phase one (line 9, Algorithm 29).

Firstly, at least $n_a - k + 1$ promises must have been received. This condition satisfies the usual revision A quorum intersection requirement (Equation 4.6). Secondly, at most one value can be a member of V_{dec}. After the first condition has been satisfied, then V_{dec} represents the set of value which maybe decided in e_{max}. A value v is only included in V_{dec} if the proposal (e_{max}, v) has been returned by sufficient acceptors that (e_{max}, v) would be decided if all remaining acceptors $(n_a - |Q_P|)$ also return the proposal (e_{max}, v).[10]

[10]This pseudocode is re-calculating V_{dec} after receiving each message, this could be done more effectively by updating V_{dec} incrementally.

Algorithm 28: Proposer algorithm with epochs by recovery and fixed quorums.

state:
- Q^e: a fixed phase two quorum for each epoch $\forall e \in E$

```
1  v, e_max ← nil
2  Q_P, Q_A ← ∅
3  if e = nil then
4      e ← 0, v ← γ, goto line 18
5  else
6      e ← e + 1
   /* Start of Phase 1 for proposal e */
7  send prepare(e) to acceptors
8  while ∃z ∈ E : z < e ∧ (e_max = nil ∨ e_max ≤ z) ∧ Q_P ∩ Q^e = ∅ do
9      switch do
10         case promise(e,f,w) received from acceptor a
11             Q_P ← Q_P ∩ {a}
12             if f ≠ nil ∧ (e_max = nil ∨ f > e_max) then
13                 e_max ← f, v ← w
14         case timeout
15             goto line 1
16 if v = nil then
17     v ← γ
   /* Start Phase 2 for proposal (e,v) */
18 send propose(e,v) to acceptors
19 while Q_A ⊉ Q^e do
20     switch do
21         case accept(e ,v) received from acceptor a
22             Q_A ← Q_A ∪ {a}
23         case timeout
24             goto line 1
25 return v
```

In the worst case, the proposals received are equally split between two values associated with the highest epochs. As such we can place the following bound on the cardinality of Q_P:

$$n_a - k + 1 \leq |Q_P| \leq 2n_a - 2k + 1$$

Table 7.1[11] shows examples of this relationship between the total number of acceptors

[11]The number of acceptors for phase one is shown for epoch 1 onwards since the proposer can always

Algorithm 29: Proposer algorithm with epochs by recovery and counting quorums.

state:
- k: size of counting quorum (configured, persistent)

```
1  e_max ← nil
2  Q_P, Q_A ← ∅
3  if e = nil then
4      e ← 0, v ← γ, goto line 24
5  else
6      e ← e + 1
7  ∀a ∈ A : R[a] ← no
8  V_dec ← ∅
   /* Start of Phase 1 for proposal e */
9  send prepare(e) to acceptors
10 while (|Q_P| ≤ n_a − k) ∨ (|V_dec| > 1) do
11     switch do
12         case promise(e,f,w) received from acceptor a
13             Q_P ← Q_P ∪ {a}
14             if f ≠ nil ∧ (e_max = nil ∨ f > e_max) then
15                 e_max ← f
16             R[a] ← (f, w)
17             V_dec ← {v ∈ V | |{a ∈ A | R[a] = (e_max, v)}| ≥ k + |Q_P| − n_a}
18         case timeout
19             goto line 1
20 if V_dec = ∅ then
21     v ← γ
22 else
23     v ← only(V_dec)
   /* Start Phase 2 for proposal (e,v) */
24 send propose(e,v) to acceptors
25 while |Q_A| < k do
26     switch do
27         case accept(e ,v) received from acceptor a
28             Q_A ← Q_A ∪ {a}
29         case timeout
30             goto line 1
31 return v
```

$\mathbf{n_a}$	$\mathbf{k}$	$\mathbf{\lvert Q_P \rvert}$
2	2	1
3	2	2 - 3
	3	1
4	3	2 - 3
	4	1
5	3	3 - 5
	4	2 - 3
	5	1
6	4	3 - 5
	5	2 - 3
	6	1
7	4	4 - 7
	5	3 - 5
	6	2 - 3
	7	1

Table 7.1: Examples of the counting quorums for epochs by recovery

(n_a), the number of acceptors for phase two (k) and the number of acceptors for phase one ($|Q_P|$).

We will now consider four possible executions of Algorithm 29. In each example, the system is comprised of 3 acceptors ($n_a = 3$), 2 proposers ($n_p = 2$) and strict majority quorums are used ($k = 2$). As before, epochs are used by proposers in sequence, starting from epoch 0. Since $e_{min} = 0$ then any proposer using it can skip phase one and proceed directly to phase two.

Firstly, we will examine Figures 7.2, 7.3 and 7.4 where the two proposers execute serially, proposer p_1 followed by proposer p_2. All three executions begin after the proposer p_1 has proposed and decided the proposal $(0, A)$. In Figure 7.2, the proposal $(0, A)$ is accepted by all acceptors. However, in Figures 7.3 and 7.4 the acceptor a_3 has not accepted the proposal $(0, A)$, due to delay/loss of the message or a slow/failed acceptor. All three cases begin with proposer p_2 proposing $(0, B)$ but p_2 does not receive the two accepts necessary to complete phase two as the value A has already been decided.

In Figure 7.2, the acceptor a_3 does not accept the proposal $(0, B)$ as it has already accepted $(0, A)$. In Figure 7.3, the acceptor a_3 is able to accept the proposal $(0, B)$ since it has not yet accepted any proposals but does not do so due to loss/failure. In Figure 7.4, the acceptor a_3 accepts the proposal $(0, B)$.

At this point in time, the three examples differ only by the state of acceptor a_3. In Figure

bypass phase one for epoch 0.

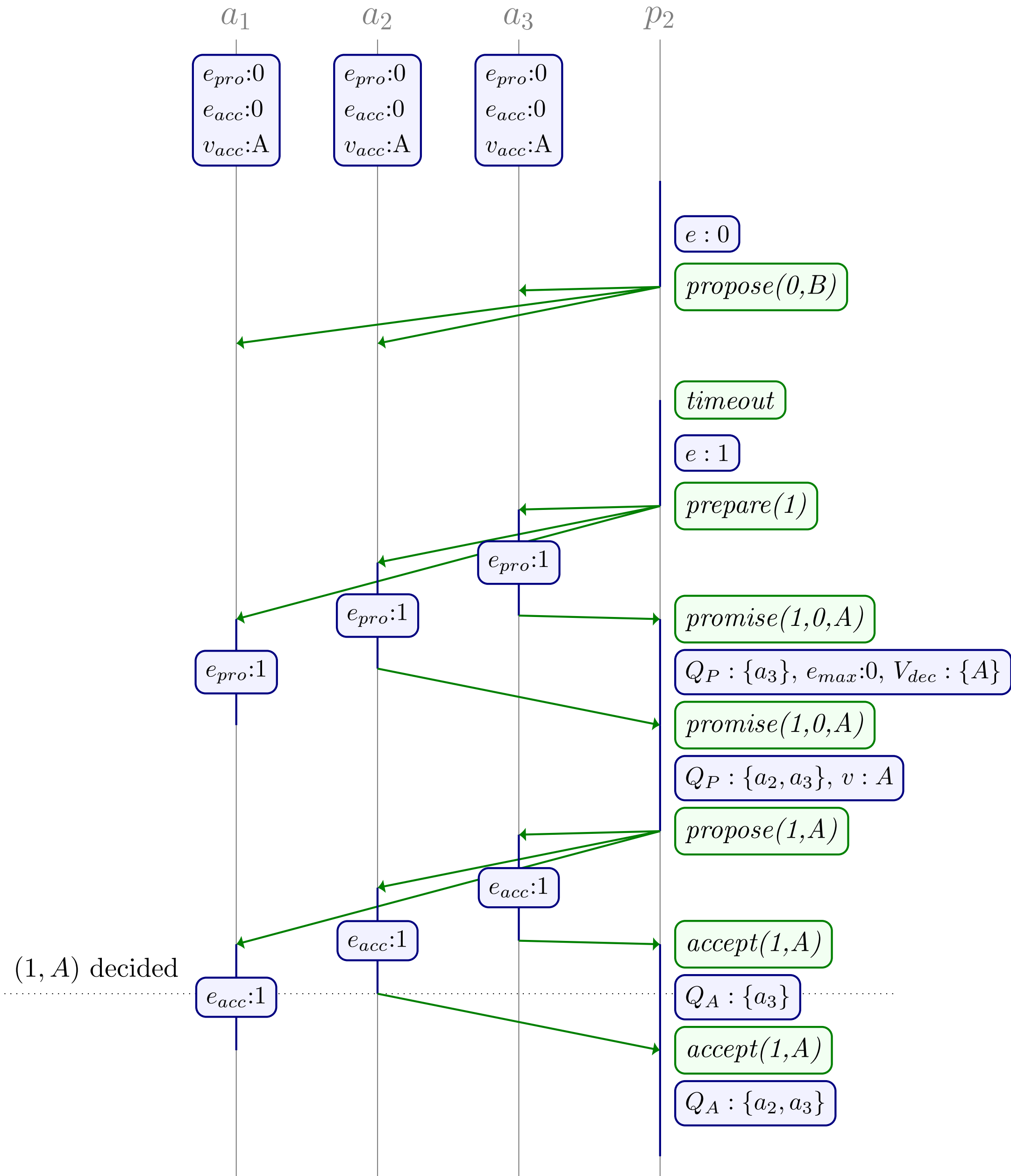

Figure 7.2: Example run of epochs by recovery with two serial proposers. The proposal $(0, A)$ was accepted by all acceptors before proposer p_2 proposes $(0, B)$.

7.2, the last accepted proposal on a_3 is $(0, A)$, in Figure 7.3, the last accepted proposal on a_3 is nil and in Figure 7.4, the last accepted proposal on a_3 is $(0, B)$. In all three examples, the proposer p_2 then retries the proposer algorithm with epoch 1 and p_2 receives promises from acceptors a_2 and a_3.

In Figure 7.2, both acceptors a_2 and a_3 return the proposal $(0, A)$ with their promises so $V_{dec} = \{A\}$ thus proposer p_2 can proceed to phase two and propose $(1, A)$.

In Figure 7.3, only acceptor a_2 returns a proposal (in this case $(0, A)$) with their promise so $V_{dec} = \{A\}$ thus proposer p_2 can proceed to phase two and propose $(1, A)$.

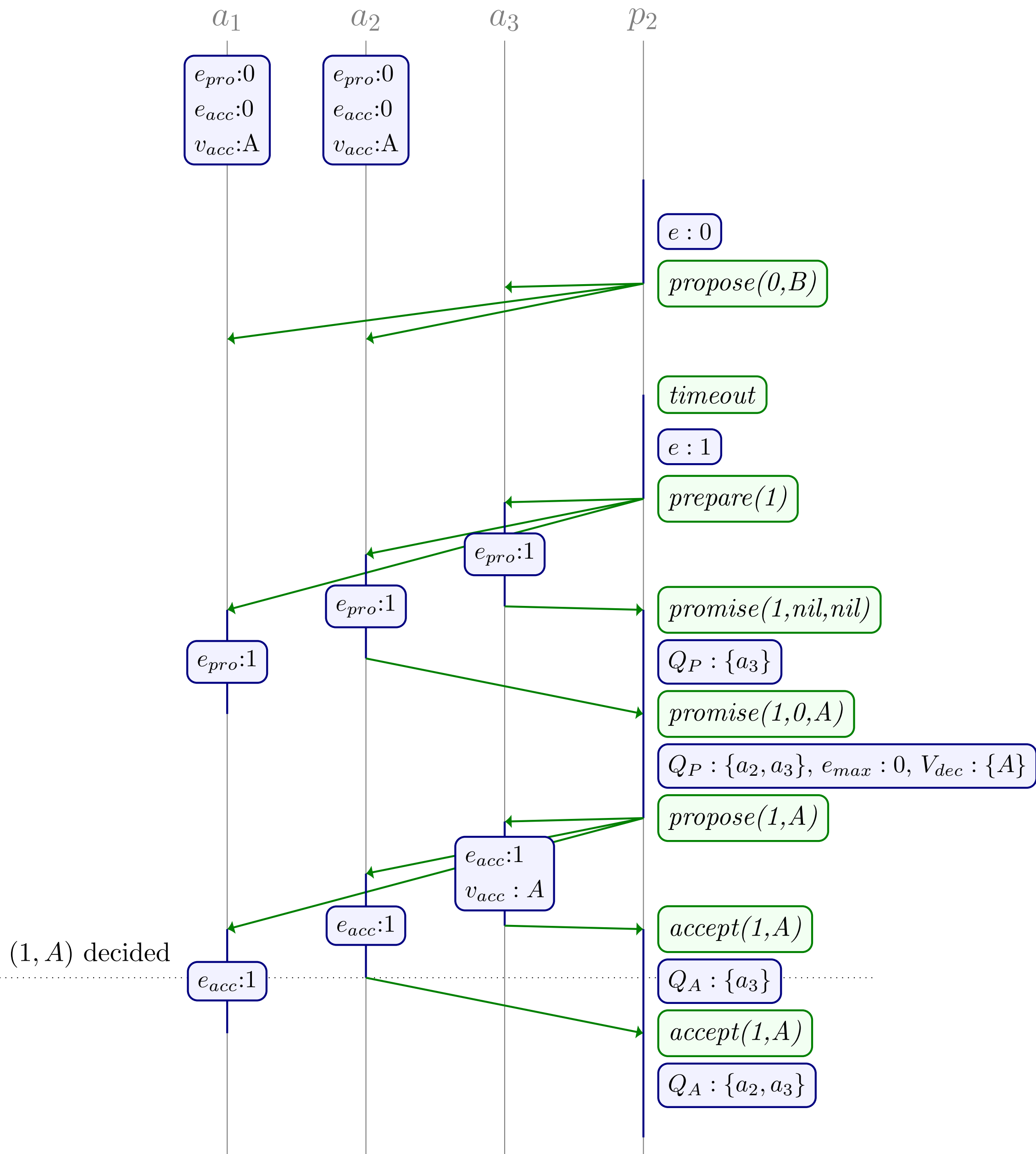

Figure 7.3: Example run of epochs by recovery with two serial proposers. Neither proposal $(0, A)$ or $(0, B)$ is accepted by acceptor a_3.

In Figure 7.4, the acceptors return two different proposals with the promises. The acceptor a_2 returns the proposal $(0, A)$ and acceptor a_3 returns the proposal $(0, B)$. At this point, $|Q_P| = 2$ and $V_{dec} = \{A, B\}$. This is a value collision thus the proposer p_2 must wait for further promises. The proposer p_2 receives the promise from acceptor a_1 with proposal $(0, A)$. It is now the case that $|Q_P| = 3$ and $V_{dec} = \{A\}$ thus p_2 is now able to proceed to phase two and propose $(1, A)$.

In contrast to earlier figures, Figure 7.5 shows two proposers executing concurrently. Both are proposing the same proposal $(0, A)$ and this proposal is quickly decided.

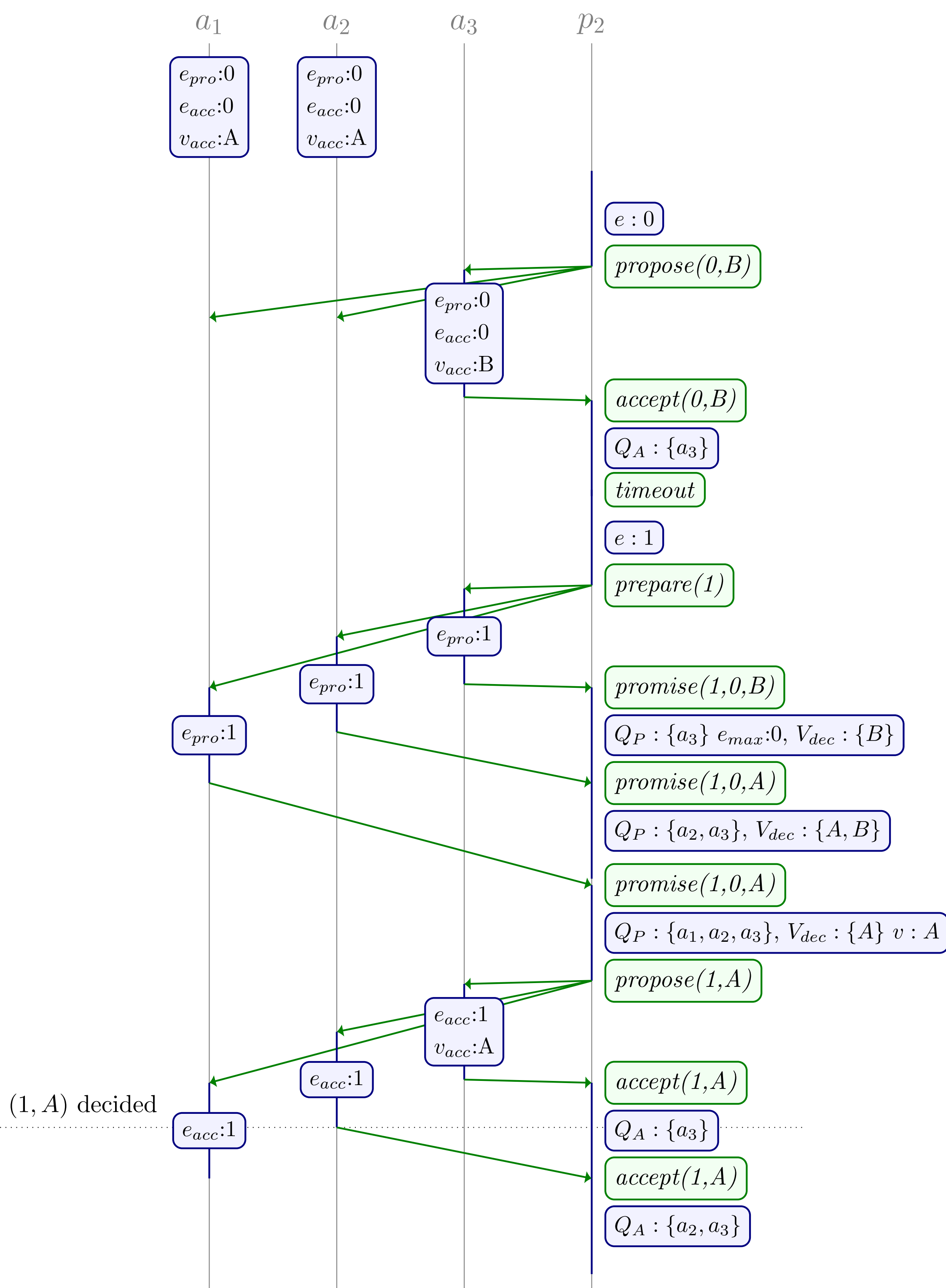

Figure 7.4: Example run of epochs by recovery with two serial proposers. The proposal $(0, B)$ is accepted by acceptor a_3.

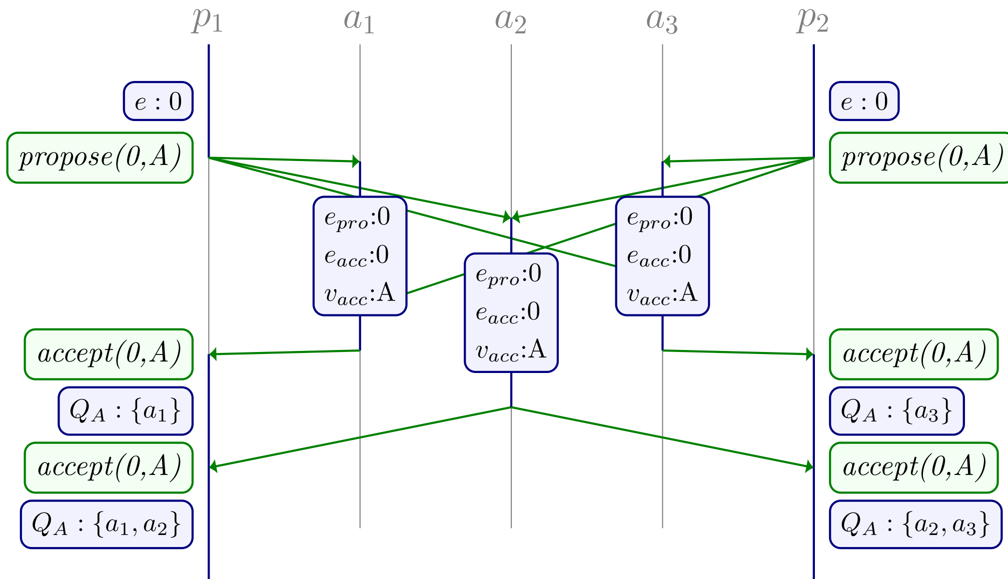

Figure 7.5: Example run of epochs by recovery with two concurrent proposers proposing the same proposal $(0, A)$.

7.4 Hybrid epoch allocation

Epoch allocation approach	Epochs unique to values	Epochs unique to proposers	Epoch assignment required
Pre-allocation	Y	Y	Y, to proposers
Voting	Y	Y	N
Allocator	Y	Y & N[12]	N
Value-based	Y	N	Y, to values
Recovery	N	N	N

Table 7.2: Approaches to epoch allocation

Thus far, we have described five mechanisms for handling the allocation of epochs: static allocation, phase one voting, dynamic allocation by an allocator (§7.1), valued-based (§7.2) or recovery-based allocation (§7.3). These mechanisms are summarised in Table 7.2[13]. However, algorithms for distributed consensus need not utilise only one of these mechanisms but may use them in combination by allocating epochs to particular methods.

The ability for proposers to use any epoch is most powerful with the epoch e_{min}, since proposers using e_{min} can skip phase one. Therefore a logical hybrid algorithm would consist of combining either an allocator, value-based or recovery-based approach for e_{min} (fast

[12]Epochs are only unique to proposers when an allocator is used if the proposal is not reallocated.
[13]Proposal copying (§3.10) can also be combined with each of these mechanisms.

path) and pre-allocation for all other epochs (slow path). We will now consider each of these three algorithms.

7.4.1 Multi-path Paxos using allocator

One of the key limitations of allocating exclusive epochs using an allocator (§7.1) is that the liveness of the system now depends on the availability of the allocator, a single participant. This can be addressed by a hybrid approach consisting of using an allocator only for e_{min} (fast path) and pre-allocation for all other epochs (slow path)[14][15].

The fast path proposer algorithm begins with a message exchange with the allocator. If the proposer is allocated e_{min} then it can bypass phase one and propose its candidate value in phase two of e_{min}.

If the fast path is unsuccessful, either because the allocator is unavailable or another proposer has already been allocated e_{min}, then the proposer executes Paxos as usual[16], this is referred to as the slow path[17].

Algorithm 30: Phase zero of Multi-path Paxos with an allocator

```
   /* Start of Phase 0 */
1  send generate-next() to allocator
2  switch do
3      case allocate(e_min) received
4          e ← e_min, v ← γ
5          goto phase two
6      case timeout or no-allocate() received
7          e ← min(ℰ)
8          ℰ ← ℰ \ {e}
9          goto phase one
10 ...
```

Algorithm 30 gives an example of *phase zero*, the epoch selection phase. If epoch e_{min} is allocated to the proposer then it proceeds to phase two. Otherwise, if e_{min} has already been allocated or the allocator does not respond, the proposer uses one of its pre-allocated epochs. The allocator could be implemented as a simple boolean flag to indicate whether e_{min} has been allocated.

[14]Equally, epochs by phase one voting could be used for all other epochs instead of pre-allocation.

[15]Note that we could extend this approach to the allocator for the first n epochs instead of just e_{min}.

[16]Except that e_{min} cannot be pre-allocated.

[17]In practice, the proposer may choose whether to first try the fast path or proceed directly to the slow path.

As described in section 7.1, we could extend the allocator to store the value associated with the epoch assigned by the service. Effectively, the allocator stores the primary copy of the value and the acceptors store the backup copy. If the allocator is available, the proposers can take the fast path. First, the proposers get/set the primary copy of the value on the allocator (phase zero) then they back up the value to a quorum of acceptors $\mathcal{Q}_2^{e_{min}}$ (phase one)[18]. Otherwise, the proposer takes the slow path, executing the classic two phase proposer algorithm over the acceptors to update the backup copies of the value.

Note, that this algorithm provides a new progress guarantee. If the system is synchronous and both the allocator and an acceptor quorum $Q \in \mathcal{Q}_2^{e_{min}}$ are live, then proposers are guaranteed to terminate in two round trips (one to the allocator and one to acceptors)[19]. This is because the allocator acts as a serialisation point, preventing duelling between proposers.

7.4.2 Multi-path Paxos using value-based allocation

Value-based allocation of epochs requires that candidate values be limited to a known range. This restriction can be weakened using Multi-path Paxos to permit values outside the known range. The first n epochs are allocated to values within the known range of size n, the most likely values should be allocated the lower epochs with the most common value assigned to e_{min}. All epochs after n are assigned to proposers by pre-allocation. If a proposer has a candidate value from the known range then the proposer can use the value assigned epoch. If this is unsuccessful or if the proposer has a candidate value outside the known range then the proposer can fall back to using the epoch assigned by pre-allocation.

As before, this algorithm provides a new progress guarantee. If all proposers are proposing the same value then they are guaranteed to terminate in two round trips (or one round trip for the value associated with the minimum epoch) even in an asynchronous system[20]. This is not the case for Classic Paxos where proposers proposing the same value could duel indefinitely.

7.4.3 Multi-path Paxos using recovery

Algorithms 31 & 32 shows an example hybrid algorithm consisting of epochs by recovery for e_{min} (fast path) and pre-allocation for all other epochs (slow path). Algorithm 31 is a fast path proposer algorithm, Algorithm 32 is the slow path proposer algorithm and the acceptor algorithm is the same as for epochs by recovery (Algorithm 24).

[18] Note that unlike the SAA, proposer cannot always read the value stored on the allocator to learn the decided value

[19] This requires the system to have been synchronous since startup.

[20] This statement assumes that NACKs are used instead of timeouts

Algorithm 31: Fast path - Proposer algorithm for Multi-path Paxos with recovery

```
   /* Start of Phase 2 for proposal (e_min,γ) */
1  Q_A ← ∅
2  send propose(e_min,γ) to acceptors
3  while |Q_A| < ⌈3n_a/4⌉ do
4      switch do
5          case accept(e_min,γ) received from acceptor a
6              Q_A ← Q_A ∪ {a}
7          case timeout
8              goto slow path
9  return γ
```

By using counting quorums of size $k = \left\lceil \frac{3n_a}{4} \right\rceil$ for e_{min} then we can use strict majority quorums for all other epochs. Such an algorithm would satisfy the same progress guarantees as Classic Paxos but with an improved best case; a decision in one round trip to $\frac{3}{4}$ of acceptors. The proposer algorithm would first try to get acceptors to accept phase two of e_{min} (fast path) and fall back to majority agreement for both phases of a subsequent epoch if unsuccessful (slow path).

We can utilise Paxos revision C to optimise algorithm 32. For all epochs e where $e \neq succ(e_{min})$, if a promise is received with a proposal (f, v) where $e = succ(f)$ then the proposer can proceed to phase two of epoch e to propose v.

Similarly, for the epoch $succ(e_{min})$, we can also proceed to phase two when at least $\left\lfloor \frac{n_a}{4} \right\rfloor + 1$ acceptors have promised; and at most one unique proposal was received with the promises.

Fast Paxos (outlined in §3.12) is a special case of Hybrid epochs, where fast epochs are shared by recovery and classic epochs are pre-allocated/voted. In Fast Paxos, all phase one quorums are of size k_c, regardless of the promises returned during phase one. This is equivalent to always waiting for the upper bound on the number of promises needed in Paxos with epochs by recovery. Thus one of the implications of epochs by recovery (other than its generality) is that phase one of Fast Paxos can be completed after fewer promises, with a minimum of $n_a - k_f + 1$ promises needed.

7.5 Summary

In this chapter, we have demonstrated various alternatives to pre-allocation or phase one voting for allocation of epochs between proposers. The methods covered included dynamic epoch allocation by an allocator, allocating epochs by value instead of by proposer and sharing epochs by recovery. These methods can be used individually or in combination.

Algorithm 32: Slow path - Proposer algorithm for Multi-path Paxos with recovery

```
1  e_max ← nil
2  Q_P, Q_A ← ∅
3  e ← min(ℰ)
4  ℰ ← ℰ \ {e}
5  ∀a ∈ A : R[a] ← no
   /* Start of Phase 1 for proposal e  */
6  send prepare(e) to acceptors
7  while |Q_P| < ⌊n_a/2⌋ + 1 do
8      switch do
9          case promise(e,f,w) received from acceptor a
10             Q_P ← Q_P ∪ {a}
11             if f ≠ nil ∧ (e_max = nil ∨ f > e_max) then
12                 e_max ← f
13             R[a] ← (f, w)
14         case timeout
15             goto line 1
16 if e_max = e_min then
17     V_dec ← {v ∈ V | |{a ∈ A|R[a] = (e_max, v)}| ≥ ⌈n_a/4⌉ }
18 else
19     V_dec ← {v ∈ V|R[_] = (e_max, v)}
20 if V_dec = ∅ then
21     v ← γ
22 else
23     v ← only(V_dec)
   /* Start Phase 2 for proposal (e,v) */
24 send propose(e,v) to acceptors
25 while |Q_A| < ⌊n_a/2⌋ + 1 do
26     switch do
27         case accept(e,v) received from acceptor a
28             Q_A ← Q_A ∪ {a}
29         case timeout
30             goto line 1
31 return v
```

Most notability, we have proposed epochs by recovery, which allows any proposer to use any epoch, provided additional intersection requirements are satisfied. Epochs by recovery generalises Fast Paxos by putting into practice our revised understanding of quorum intersection (§4.2) and value selection (§6). Any proposer may decide a value in one round trip compared to Classic Paxos in which any proposer may decide a value in two trips or Multi-Paxos which allowed one proposer, the leader, to decide a value in one round trip.

Our motivation for re-examining how epochs are allocated was to overcome the limitations of exclusive epoch allocation, particularly that only one proposer may utilise the minimum epoch to bypass phase one. In pursuit of this goal, we have also found that these techniques can provide stronger progress guarantees in particular scenarios, sometimes these guarantees even held under weaker assumptions. For example, in epochs by recovery, multiple proposers proposing the same value may not duel and will terminate in one round trip, without assuming synchrony[21].

[21] This statement assumes we are using NACKs instead of timeouts.

Chapter 8

Conclusion

> The most useful piece of learning for the uses of life is to unlearn what is untrue.
>
> *Antisthenes*

Paxos has been synonymous with distributed consensus for over two decades. As such, it has been extensively researched, taught and deployed in production. This thesis sought to reconsider how we approach consensus in distributed systems and challenge the widely held belief that the Paxos algorithm is an optimal solution to consensus.

8.1 Motivation

In section 1.3, we outline limitations of Paxos. Aside from the algorithm's subtlety and underspecification, decisions are slow as two round trips to the majority of acceptors is needed for each decision. This approach leads to a high message overhead, which increases linearly with the number of acceptors and is limited in scalability as each additional acceptor increases the size of the majority and thus decreases performance. Paxos relies on synchrony to avoid duelling between proposers and also relies on the majority of acceptors being live to make progress.

Paxos tightly couples the number of participants, availability in the face of failures and steady state performance. Paxos offers a one-size-fits-all solution to distributed consensus which is highly symmetric, following a single set execution path, regardless of the state of the system. Paxos guarantees that a proposer will terminate in two rounds given its liveness conditions, namely synchrony, exactly one proposer is live and at least a majority of acceptors are live. If these conditions are not satisfied, Paxos provides little in the way of progress guarantees. If stronger conditions are satisfied then Paxos still requires two rounds with majority agreement to make progress.

In practice, when reaching agreement over a sequence with Paxos, the Multi-Paxos optimisation is used almost exclusively. So much so that the terms Paxos and Multi-Paxos are often used interchangeably. The academic literature has proposed agreeing upon a sequence without Multi-Paxos, for example using Fast Paxos, however, such proposals have seen little practical application. Multi-Paxos allows agreements to be reached in one round trip to the majority of acceptors, ignoring the possible extra round trip to the leader and back. Whats more, the leader in Multi-Paxos acts a point of serialisation, preventing duelling between proposers, however, synchrony is needed to reliably detect and replace failed leaders. The primary limitation of centralised approaches such as Multi-Paxos is that the leader is the performance bottleneck.

8.2 Summary of contributions

This thesis proves that Paxos is conservative in its approach by weakening the requirements for quorum intersection, phase completion, value selection and epoch allocation.

After outlining the widely known Classic Paxos algorithm in chapter 2, we begin with our systemisation of knowledge study (chapter 3) which surveyed the key refinements to the Classic Paxos algorithm.

In chapter 4, we revised the Paxos's quorum intersection from:

$$\forall Q, Q' \in \mathcal{Q} : Q \cap Q' \neq \emptyset$$

to the following, for each epoch e:

$$\forall Q \in \mathcal{Q}_1^e, \forall f \in E : f < e \implies \forall Q' \in \mathcal{Q}_2^f : Q \cap Q' \neq \emptyset$$

In other words, we have shown that it is not necessary to require that phase one quorums intersect, nor that phase two quorums intersect nor that the phase one quorum intersects with phase two quorums of subsequent epochs.

In chapter 5, we proved that if a proposer received a promise with the proposal (e, v) then this is sufficient to satisfy the quorum intersection requirements for epochs up to e (inclusive).

In chapter 6, we demonstrated that Paxos's value selection rule of proposing the value associated with the greatest epoch is a conservative approximation of quorum based value selection. If more promises are received than is necessary to satisfy quorum intersection then tracking quorums can allow a proposer to propose their candidate values, instead of being required to propose a previous value.

These revisions of quorum intersection, phase completion and value selection come together in section 7.3, when we remove the requirement that epochs are unique to proposals. The technique, referred to as epochs by recovery, generalises over the Fast Paxos algorithm by weakening its quorum intersection requirements. Furthermore, it applies our quorum-based value selection method to allow proposers to complete phase one with fewer promises and provides greater flexibility over the value proposed.

We also proposed various alternatives to epoch allocation by recovery, such as epochs from an allocator (§7.1) or epochs by value (§7.2). These can be used instead of or together with the existing epoch allocation methods.

8.3 Implications of contributions

Over the course of this thesis, we have proposed a generalised algorithm for solving distributed consensus, a powerful primitive for architecting distributed systems. In §1.4, we proposed the following two research questions:

Are the limitations of Paxos inherent to the problem of consensus or specific to the approach taken by the Paxos algorithm?

Is the Paxos algorithm the optimal solution to distributed consensus?

We believe that we have improved over the Paxos algorithm and demonstrated that some of its limitations are specific to its approach. We now discuss this further, divided into four domains, greater flexibility, new progress guarantees, improved performance and better clarity.

8.3.1 Greater flexibility

The algorithm we have proposed is no 'silver bullet' solution. Instead, it is a flexible family of approaches for constructing a broad spectrum of consensus algorithms, suitable for many deployment environments, optimised for different workloads and offering new tradeoffs in performance and reliability. The breadth of algorithms proposed aims to reflect the diverse landscape of today's distributed systems. The algorithms we have proposed in this thesis introduce asymmetry to Paxos, offering multiple pathways for proposers to reach termination, varying depending on the state of the system.

We began by using our weakened quorum intersection requirements to introduce the notion of varying quorums by epochs. For example, in §4.2, we presented *All aboard Paxos*, which provided two routes for proposers (co-located with acceptors) to terminate:

- termination in one round trip to all acceptors using epochs 0 to k; or
- termination in two rounds trips to a majority of acceptors using epochs from $k + 1$.

Likewise, we see another example of this multi-pathway approach in Paxos revision C (Chapter 5), where a proposer in phase one of epoch e can satisfy its intersection requirement with the phase two of a previous epoch f either:

- by receiving promises from at least one acceptor in each quorum $Q \in Q_2^f$; or
- by receiving a promise with a proposal from an epoch g where $f \leq g \leq e$.

In section 3.10, we allow proposers the option to copy an existing proposal instead of starting a new proposal. In section 7.4, we propose a hybrid approach consisting of using epoch allocation by an allocator, value mapping or recovery for the minimum epoch and epoch allocation by pre-allocation or voting for all other epochs.

8.3.2 New progress guarantees

Paxos focuses on a single progress property: guaranteed progress regardless of an algorithm's current state. Whilst useful for comparing fault-tolerance of algorithms under worst case conditions, this gives us little information regarding the overall availability of the algorithms. Over the course of this thesis, we have demonstrated algorithms with new progress properties depending on the system state. In this section, we will consider various examples.

A proposer can terminate in one round trip to a phase two quorum of acceptors if phase one has been completed and no acceptors in the quorum promise or accept since phase one (§4.1). At the extreme, this quorum may only contain only one acceptor, as described in §4.3.2. The tradeoff for optimising the phase two quorum is decreased performance and availability for the phase one quorum. This tradeoff may be worthwhile when combined with Multi-Paxos, which seldom executes phase one compared to phase two (§4.3.3).

A proposer can terminate in one round trip to a phase two quorum of acceptors if it is the first to propose and has been allocated e_{min}, since this proposer is able to bypass phase one. More generally, since each proposer during phase one is required to intersect only with the phase two quorums of previous epochs, the intersection requirements build up as epochs increase (§4.2.3).

Our progress guarantee for Classic Paxos relies on a single proposer executing the proposer algorithm. In practice, this is often achieved by designating one proposer as distinguished and thus relying on synchrony to detect failure of the designated proposer.

In sections 7.2 and 7.3, we proposed the allocation of epochs to values and epochs allocation by recovery. Both these new algorithms can guarantee termination when multiple proposers are executing the proposer algorithm with the same candidate value. For example, this was illustrated in Figure 7.5.

In section 7.4, we proposed a hybrid algorithm consisting of using an allocator for the minimum epoch and pre-allocation for all other epochs. Provided the allocator is live and synchronous, then any number of proposers will terminate in two rounds trips (one to the allocator, one to the acceptors).

8.3.3 Improved performance

Our generalisations provide the opportunity to improve the best case performance and/or to increase the likelihood of the best case occurring in practice. Optimising for the steady state has allowed us to improve overall performance. The tradeoff for this gain can be decreased performance during the rarer failure case behaviour. Unlike Classic Paxos, we do not enforce a particular tradeoff between performance and availability. Instead, this tradeoff is an application specific decision.

This is best illustrated by combining Multi-Paxos with the weakened quorum intersection between phases (§4.1). We can now choose our tradeoff between phase one quorums, which are rarely used as they are only needed when replacing a leader and phase two quorums, which are used for every decision.

The key motivation for Multi-Paxos is to reach agreement in one round trip, however, its centralised approach is a substantial performance bottleneck. We have proposed various other mechanisms to achieve one round trip agreement without centralisation, including the following:

If proposers and acceptors are co-located on each participant:

- A participant can execute phase one locally, provided it uses all participants for phase two (§4.1).
- A participant can complete phase one locally, provided the participant has accepted a proposal from the predecessor epoch (§5).

Otherwise:

- One of the proposers can skip phase one if it has been allocated of the minimum epoch (§4.2.3). When reaching agreement over a sequence, this proposer can be rotated to avoid centralisation.
- A proposer can skip phase one to propose its candidate value if its candidate value is assigned to the minimum epoch using epoch allocation by values (§7.2)
- Any proposer can skip phase one to propose its candidate value if the minimum epoch is assigned by epoch allocation by recovery (§7.3)

8.3.4 Better clarity

At the very least, we hope to have furthered understanding of this important and surprisingly subtle field of distributed systems.

Bibliography

[ACDK17] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tevfik Kosar. Multileader WAN paxos: Ruling the archipelago with fast consensus, 2017. arXiv:1703.08905 [cs.DC].

[BAC+13] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's distributed data store for the social graph. In *Proceedings of the 2013 USENIX Annual Technical Conference*, ATC'13, pages 49–60, Berkeley, CA, USA, 2013. USENIX Association.

[BBH+11] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. Paxos replicated state machines as the basis of a high-performance data store. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 141–154, Berkeley, CA, USA, 2011. USENIX Association.

[Bir85] Kenneth P. Birman. Replication and fault-tolerance in the ISIS system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, SOSP '85, pages 79–86, New York, NY, USA, 1985. ACM.

[BJ87] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems (TOCS)*, 5(1):47–76, January 1987.

[Bur06] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4:1–4:26, June 2008.

[CGR07] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM.

[CHT96] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM (JACM)*, 43(4):685–722, July 1996.

[CL99] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, March 1996.

[DDS87] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM (JACM)*, 34(1):77–97, January 1987.

[Dem] Murat Demirbas. Modeling Paxos and Flexible Paxos in Pluscal and TLA+. `http://muratbuffalo.blogspot.co.uk/2016/11/modeling-paxos-and-flexible-paxos-in.html`. [Online; accessed 17-Jan-2018].

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[DLS88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, April 1988.

[FLP85] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, April 1985.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[GL03] Eli Gafni and Leslie Lamport. Disk Paxos. *Distributed Computing*, 16(1):1–20, February 2003.

[HKJR10] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference*, ATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[JRS11] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 41st IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, pages 245–256. IEEE, 2011.

[Lam78a] Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2(2):95 – 114, August 1978.

[Lam78b] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM (CACM)*, 21(7):558–565, July 1978.

[Lam96] Butler W. Lampson. How to build a highly available system using consensus. In *Proceedings of the 10th International Workshop on Distributed Algorithms*, WDAG '96, pages 1–17, London, UK, UK, 1996. Springer-Verlag.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, May 1998.

[Lam01a] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)*, December 2001.

[Lam01b] Butler Lampson. The ABCD's of Paxos. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, PODC '01, pages 13–, New York, NY, USA, 2001. ACM.

[Lam05a] Leslie Lamport. Fast Paxos. Technical Report MSR-TR-2005-112, Microsoft Research, 2005.

[Lam05b] Leslie Lamport. Generalized consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, March 2005.

[LC12] Barbara Liskov and James Cowling. Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT, July 2012.

[LM04] Leslie Lamport and Mike Massa. Cheap Paxos. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, DSN '04, pages 307–, Washington, DC, USA, 2004. IEEE Computer Society.

[LVA+15] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. Existential consistency: Measuring and understanding consistency at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 295–310, New York, NY, USA, 2015. ACM.

[MAK13] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 358–372, New York, NY, USA, 2013. ACM.

[Mal] Dahlia Malkhi. ACM A.M. Turing award - Leslie Lamport 2013. `https://amturing.acm.org/award_winners/lamport_1205376.cfm`. [Online; accessed 23-April-2018].

[Mel17] Max Meldrum. Flexible Paxos: An industry perspective. Master's thesis, Blekinge Institute of Technology, 2017.

[MHL+92] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)*, 17(1):94–162, March 1992.

[MJM08] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machines for WANs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 369–384, Berkeley, CA, USA, 2008. USENIX Association.

[MLZ08] Dahlia Malkhi, Leslie Lamport, and Lidong Zhou. Stoppable Paxos. Technical Report MSR-TR-2008-192, Microsoft Research, April 2008.

[MOZ05] Dahlia Malkhi, Florin Oprea, and Lidong Zhou. Omega meets Paxos: Leader election and stability without eventual timely links. In *Proceedings of the 19th International Conference on Distributed Computing*, DISC'05, pages 199–213, Berlin, Heidelberg, 2005. Springer-Verlag.

[MPP12] P.J. Marandi, M. Primi, and F. Pedone. Multi-ring Paxos. In *Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12, June 2012.

[MPSP10] P.J. Marandi, M. Primi, N. Schiper, and F. Pedone. Ring Paxos: A high-throughput atomic broadcast protocol. In *Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 527–536, June 2010.

[NAEA18] 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*, SIGMOD '18, pages 1221–1236, New York, NY, USA, 2018. ACM.

[OL88] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 8–17, New York, NY, USA, 1988. ACM.

[OO14] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference*, ATC'14, pages 305–320, 2014.

[PLL97] Roberto De Prisco, Butler W. Lampson, and Nancy A. Lynch. Revisiting the Paxos algorithm. In *Proceedings of the 11th International Workshop on Distributed Algorithms*, WDAG '97, pages 111–125, London, UK, UK, 1997. Springer-Verlag.

[PLSS17] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos made EPR: Decidable reasoning about distributed protocols. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):108:1–108:31, October 2017.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, December 1990.

[Tre] Trex. An embeddable paxos engine for the JVM. `https://github.com/trex-paxos/trex`. [Online; accessed 17-Jan-2018].

[VRA15] Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Computing Surveys (CSUR)*, 47(3):42:1–42:36, February 2015.

[vRSS15] R. van Renesse, N. Schiper, and F. B. Schneider. Vive la difference: Paxos vs. Viewstamped Replication vs. Zab. *IEEE Transactions on Dependable and Secure Computing*, 12(4):472–484, July 2015.