Relaxed Paxos: Quorum Intersection Revisited (Again)

Heidi Howard
heidi.howard@cl.cam.ac.uk
University of Cambridge

Richard Mortier
richard.mortier@cl.cam.ac.uk
University of Cambridge

Abstract

Distributed consensus, the ability to reach agreement in the face of failures, is a fundamental primitive for constructing reliable distributed systems. The Paxos algorithm is synonymous with consensus and widely utilized in production. Paxos uses two phases: phase one and phase two, each requiring a quorum of acceptors, to reach consensus during a round of the protocol. Traditionally, Paxos requires that all quorums, regardless of phase or round, intersect and majorities are often used for this purpose. Flexible Paxos proved that it is only necessary for phase one quorum of a given round to intersect with the phase two quorums of all previous rounds.

In this paper, we re-examine how Paxos approaches the problem of consensus. We look again at quorum intersection in Flexible Paxos and observe that quorum intersection can be safely weakened further. Most notably, we observe that if a proposer learns that a value was proposed in some previous round then its phase one no longer needs to intersect with the phase two quorums from that round or from any previous rounds. Furthermore, in order to provide an intuitive explanation of our results, we propose a novel abstraction for reasoning about Paxos which utilizes write-once registers.

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

Keywords: Distributed Consensus, Paxos

ACM Reference Format:
Heidi Howard and Richard Mortier. 2022. Relaxed Paxos: Quorum Intersection Revisited (Again). In *9th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC'22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3517209.3524040

PaPoC'22, April 5–8, 2022, RENNES, France

ACM ISBN 978-1-4503-9256-3/22/04.
https://doi.org/10.1145/3517209.3524040

1 Introduction

We depend upon distributed systems, yet the computers and networks that make up these systems are asynchronous and unreliable. The longstanding problem of distributed consensus formalizes how to reliably reach agreement in such systems. When solved, we become able to construct strongly consistent distributed systems from unreliable components [19].

The Paxos algorithm [12] is widely deployed in production to solve distributed consensus [2–4]. Despite its popularity, Paxos is notoriously difficult to understand, leading to much follow up work, explaining the algorithm in simpler terms [1, 7, 13–18, 21].

Paxos operates over a series of *rounds* (sometimes referred to as proposal numbers or ballot numbers). In each round, a *proposer* may attempt to decide a value by executing a two phase protocol. Each phase of the protocol requires agreement from a quorum of *acceptors*. All quorums are required to intersect, regardless of round or phase. In phase one of Paxos, the proposer learns which values have been *accepted* in previous rounds. In phase two of Paxos, the proposer proposes a value in the current round. If the proposer learned in phase one that one or more values had been previously accepted then it must propose the value with the greatest round. If a quorum of acceptors accepts the proposal in phase two then the proposed value is decided.

Our understanding of Paxos is ever-evolving, for example, Flexible Paxos [10] observed that intersection is not needed between all quorums in Paxos. Specifically, the authors observed that it is only necessary for phase one quorums of a given round to intersect with the phase two quorums of all previous rounds. If the same quorum system is used for all rounds, then this can be simplified to the statement that only phase one and phase two quorums must intersect.

This paper re-examines how Paxos approaches the problem of consensus with the aim of further weakening the quorum intersection requirements of Paxos and improving understanding of this famously difficult algorithm. In recent years, immutability has been utilized in distributed systems to tame complexity [8, 20].

The success of these efforts has inspired us to apply immutability to the problem of distributed consensus. We proceed as follows. Once we have defined consensus (§2), we propose an abstract solution to consensus that uses only write-once registers to enable more intuitive reasoning about

safety (§3). Using the abstractions developed so far, we then describe a novel variant of Paxos, known as *Relaxed Paxos* (§4), which generalizes over Paxos and Flexible Paxos. We compare our Relaxed Paxos protocol to the original Paxos algorithm (§5) and observe the following: If a proposer learns that a value was proposed in some previous round then it is no longer required to hear from at least one acceptor in each quorum for both that round or from all previous rounds to complete phase one.

2 Problem definition

The classic formulation of single-degree consensus considers how to decide upon a single value in a distributed system. This seemingly simple problem is made non-trivial by the weak assumptions made about the underlying system: we assume only that the algorithm is correctly executed (i.e., the non-Byzantine model). We do not assume that participants are reliable. For safety, we do not assume that the system is synchronous, participants may operate at arbitrary speeds and messages may be arbitrarily delayed.

We consider systems comprised of two types of participants: *acceptors*, which store the value, and *proposers*, which read/write the value. Proposers take as input a value to be proposed and produce as output the value decided by the acceptors. Messages may only be exchanged between proposers and acceptors and we assume that the set of participants, acceptors and proposers, is fixed and known to the proposers.

An algorithm, such as Paxos, solves consensus if it satisfies the following three requirements:

Non-triviality All output values must have been the input value of a proposer.

Agreement All proposers that output a value must output the same value.

Progress All proposers must eventually output a value.

As termination cannot be guaranteed in an asynchronous system where failures may occur [6], consensus algorithms need only guarantee progress assuming partial synchrony [5].

If we have only one acceptor, then solving consensus is straightforward. Assume the acceptor has a single write-once register, r_0, to store the decided value. A *write-once register* is a persistent variable that once written cannot be modified. Proposers send requests to the acceptor with their input value. If r_0 is unwritten, the value received is written to r_0 and is returned to the proposer. If r_0 is already written, then the value in register r_0 is read and returned to the proposer. The proposer then outputs the returned value. This algorithm achieves consensus but requires the acceptor to be available for proposers to terminate. Overcoming this limitation requires the deployment of more than one acceptor, so we now consider how to generalize to multiple acceptors.

i	Q_i
$0, 1, \ldots$	$\{\{a_0, a_1\}, \{a_0, a_2\}, \{a_1, a_2\}\}$

(a) Majority quorums over 3 acceptors ($\{a_0, a_1, a_2\}$).

i	Q_i
0	$\{\{a_0, a_1, a_2\}\}$
$1, 2, \ldots$	$\{\{a_0, a_1\}, \{a_0, a_2\}, \{a_1, a_2\}\}$

(b) Quorums can vary by round. Round 0 uses all 3 acceptors, round 1 onwards uses majority quorums.

i	Q_i
$0, 2, \ldots$	$\{\{a_0, a_1\}\}$
$1, 3, \ldots$	$\{\{a_2, a_3\}\}$

(c) Quorums do not need to intersect. Even rounds use two acceptors and odd rounds using the other two acceptors.

Figure 1. Sample quorum configurations.

	a_0	a_1	a_2
r_0	⊥	⊥	B
r_1	⊥	⊥	⊥
r_2		A	A

(a) No decision in round 0 and 1. Value A decided in round 2

	a_0	a_1	a_2
r_0	⊥	A	A
r_1	A	A	

(b) Value A decided in round 0 and round 1

Figure 2. Sample state tables for a system using majority quorums (Figure 1a).

3 Abstract solution to distributed consensus

Consider a finite set of acceptors, $\{a_0, a_1, \ldots, a_n\}$, where each acceptor has an infinite series of write-once registers, $\{r_0, r_1, \ldots\}$. At any time, each register is in one of the three states:

- **unwritten**, the starting state for all registers;
- **contains a value**, e.g., A, B, C; or
- **contains *nil***, a special value denoted as ⊥.

A quorum, Q, is a non-empty subset of acceptors, such that if all acceptors have the same (non-nil) value v in the same register r_i then value v is said to be *decided*. The state of each *round*, $i \in \mathbb{N}_0$, is the set comprised of the register r_i from each acceptor. Each round i is configured with a set of quorums, Q_i, and some examples are given in Figure 1. The state of all registers across the acceptors can be represented in a table, known as a *state table*, where each column represents the state of one acceptor and each row represents a register. By combining a configuration with a state table, we can determine whether any decision(s) have been reached, as shown in Figure 2.

Figure 3 describes an abstract solution to consensus by giving four rules governing how proposers interact with registers to ensure that the safety requirements (non-triviality

Rule 1: Quorum agreement. A proposer may only output a value v if it has read v from register r_i on a quorum of acceptors $Q \in \mathcal{Q}_i$.
Rule 2: New value. A proposer may only write a (non-nil) value v provided that either v is the proposer's input value or that the proposer has read v from a register.
Rule 3: Current decision. A proposer may only write a (non-nil) value v to register r_i provided that no value v' where $v \neq v'$ can also be decided in round i.
Rule 4: Previous decisions. A proposer may only write a (non-nil) value v to register r_i provided no value v' where $v \neq v'$ can be decided in rounds 0 to $i-1$.

Figure 3. The four rules for correctness.

and agreement) for consensus are satisfied. See Appendix A for a proof.

Rule 1 (*quorum agreement*) ensures that proposers only output values that have been decided. Rule 2 (*new value*) ensures that only proposer input values can be written to registers thus only proposer input values can be decided and output by proposers. Rules 3 and 4 ensure that no two quorums can decide upon different values. Rule 3 (*current decision*) ensures that all decisions made in a round will be for the same value whilst Rule 4 (*previous decisions*) ensures that all decisions made by different rounds are for the same value.

Note that none of the four rules restrict when a proposer can write *nil* ($\perp$) to a register. The *nil* value ensures that proposers can always safely write to any register. This allows proposers to block quorums from making decisions if the proposer is unable to utilize the quorum due to Rule 3 or 4. The *nil* value is not necessary for safety, however, we will see later on how algorithms such as Paxos can utilize *nil* to satisfy the progress requirement of consensus (§4).

Rules 1 and 2 are easy to implement, but Rules 3 and 4 require more careful treatment.

3.1 Satisfying rule three

We can satisfy Rule 3 (*current decision*) if we require that all (non-nil) values written to a given round are the same. This can be achieved by assigning rounds to proposers in a round-robin fashion (for instance, with three proposers, p_0 can write to round 0, 3, 6, .. and p_1 can write to round 1, 4, 7, .. and so on) and requiring that proposers write at most one (non-nil) value to each of their own rounds. This approach ensures that at most one (non-nil) value is ever written to each round and therefore at most one value can be decided by each round.

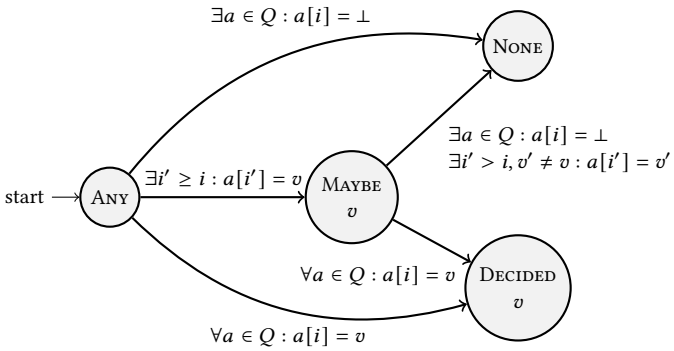

Figure 4. Computing the decisions state for quorum $Q \in \mathcal{Q}_i$ over round i. We use $a[i] = v$ as shorthand for when a proposer has read value v from register r_i on acceptor a.

3.2 Satisfying rule four

Rule 4 (*previous decisions*) requires proposers to ensure that, before writing a (non-nil) value, previous rounds cannot decide a different value. This is trivially satisfied for round 0, however, more work is required by proposers to satisfy this rule for subsequent rounds.

Assume each proposer maintains its own local copy of the state table. Initially, each proposer's state table is empty as they have not yet learned anything regarding the state of the acceptors. A proposer can populate its state tables by reading registers and storing the results in its copy of the state table. Since the registers are persistent and write-once, if a register contains a value (nil or otherwise) then any reads will always remain valid. Each proposer's state tables will therefore always contain a subset of the values from the state table.

From its local state table, each proposer can track whether decisions have been reached or could be reached by previous quorums, using a *decision table*. At any given time, each quorum is in one of four decision states:

Any: Any value could be decided by this quorum.
Maybe *v*: If this quorum reaches a decision, then value v will be decided.
Decided *v*: The value v has been decided by this quorum; a final state.
None: This quorum will not decide a value; a final state.

The rules for updating the decision table are detailed below and in Figure 4. Initially, the decision state of all quorums is Any. If there is a quorum where all registers contain the same (non-nil) value v then its decision state is Decided v. When a proposer reads *nil* from register r_i on acceptor a then for all quorums $Q \in \mathcal{Q}_i$ where $a \in Q$, the decision state Any/Maybe v becomes None. When a proposer reads a non-*nil* value v from a register r_i then for all quorums over rounds 0 to i, the decision state Any becomes Maybe v and Maybe v' where $v \neq v'$ becomes None. These rules use the

Rule 1: Quorum agreement. A proposer may output value v provided at least one quorum state is Decided v.
Rule 2: New value. A proposer p may write a non-nil value v to a register provided v is p's input value or has been read from a register.
Rule 3: Current decision. A proposer p may write a non-nil value v to a register r_i provided round i has been allocated to p but not yet used.
Rule 4: Previous decisions. A proposer may write a non-nil value v to register r_i provided the decision state of each quorum from rounds 0 to $i-1$ is None, Maybe v or Decided v.

Figure 5. The four rules for correctness using proposer decision tables.

knowledge that if a proposer reads a (non-nil) value v from the register r_i on acceptor a, it learns that all quorums in the round i must decide v if they reach a decision (Rule 3), and if any quorum of rounds 0 to $i-1$ reaches a decision then value v is decided (Rule 4).

Figure 5 describes how proposers can use state tables and decision tables to implement all four rules for correctness (Fig. 3).

4 Relaxed Paxos

We now describe Paxos [12] in terms of decision tables. We refer to our description of Paxos as *Relaxed Paxos* to distinguish it from the usual descriptions of Paxos. Like Paxos, Relaxed Paxos consists of two phases: phase one and phase two. In phase one, after choosing a round i, a proposer reads from rounds 0 to $i-1$ until it learns which value v is safe to write to round i. In phase two, the proposer writes the value v to round i and it outputs v once it learns that a quorum of acceptors has value v in register r_i.

We now consider each phase of Relaxed Paxos in more detail. Note that a proposer can complete phase one as soon as the completion criteria (underlined) has been satisfied.
Phase One

- A proposer p chooses its next round i. Once the decision state of all quorums from rounds 0 to $i-1$ is None or Maybe v then the proposer p chooses the value v (or if all states are None then its input value) and proceeds to phase two.
- The proposer p sends $\langle$P1a, $i\rangle$ to all acceptors.
- Upon receiving $\langle$P1a, $i\rangle$, each acceptor checks if register r_i is unwritten. If so, any unwritten registers up to r_{i-1} (inclusive) are set to *nil*. The acceptor replies with $\langle$P1b, $i, \mathcal{R}\rangle$ where $\mathcal{R}$ is a set of all written registers.
- Each time the proposer p receives a P1b, it updates its state and decision tables accordingly. If the proposer p times out before completing phase one, it restarts phase one with a greater round.

Phase Two

- The proposer p sends $\langle$P2a, $i, v\rangle$ to all acceptors where i is the round chosen at the start of phase one and v is the value chosen at the end of phase one.
- Upon receiving $\langle$P2a, $i, v\rangle$, each acceptor checks if register r_i is unwritten. If so, any unwritten registers up to r_{i-1} (inclusive) are set to *nil* and register r_i is set to the value v. The acceptor replies with $\langle$P2b, $i, v\rangle$.
- Each time the proposer p receives a P2a, it updates its state and decision tables accordingly. Once the decision state of a quorum is Decided v then the proposer p outputs the value v. If the proposer p times out before completing phase two, it restarts phase one with a greater round.

Once a proposer outputs the decided value, its state and decision tables are no longer needed. Optionally, a P3a can be used to notify the acceptors of the decided value, which can be recorded and the registers garbage collected.

Note that when a proposer restarts Relaxed Paxos after timing out, it does not need to clear its state table and decision table. Instead, the proposer can safely reuse what it learned about the state of acceptors in earlier executions.

Similarly, the proposer can also update its decision table for its previously assigned rounds. For instance, if it did not write a value in an assigned round then the decision state is for all quorums in that round is None.

Phase two implements Rule 1 by requiring a proposer to wait until it has a quorum in its decision table with decision state Decided v before it outputs v. Relaxed Paxos ensures Rule 2 as a proposer will only write a (non-nil) value v if v is from a decision state Maybe v or if v is the proposer's input value. Each proposer will only execute Relaxed Paxos at most once for each assigned round, thus ensuring Rule 3. The purpose of phase one is to implement Rule 4, by requiring that a proposer does not write any (non-nil) value v to round i until it has checked that all previous quorums (over rounds 0 to $i-1$) either cannot reach a decision or will decide the same value v. Before an acceptor sends a P1b in round i, it writes *nil* to any unwritten registers from r_0 to r_{i-1}, effectively blocking previous rounds from deciding new values.

Figure 6 gives an example of the message exchange as two proposers execute Relaxed Paxos with three acceptors.

5 Implications for Paxos

Relaxed Paxos differs from the usual descriptions of Paxos as it encapsulates various generalizations regarding quorums.

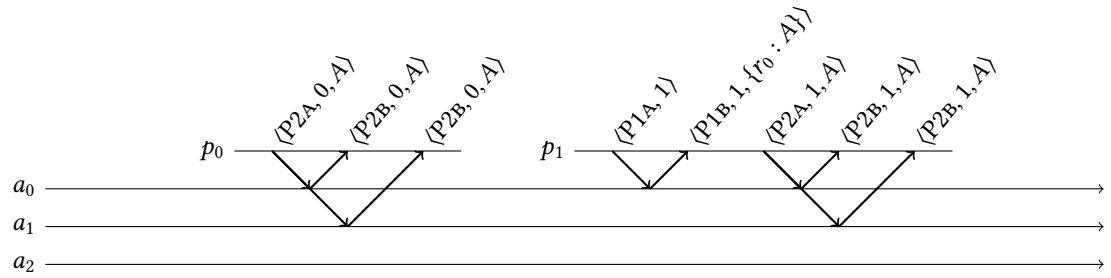

Figure 6. Sample message exchange for Relaxed Paxos between two proposers ($\{p_0, p_1\}$) and three acceptors ($\{a_0, a_1, a_2\}$). Majority quorums (Figure 1a) are used. Some messages are omitted for simplicity. Corresponding examples of the proposer's state table and decision table are given in Figure 7 for proposer p_0 and Figure 8 for proposer p_1.

	a_0	a_1	a_2
r_0			

i	Q	Decision
0	$\{a_0, a_1\}$	Any
0	$\{a_0, a_2\}$	Any
0	$\{a_1, a_2\}$	Any

(a) Initial state.

	a_0	a_1	a_2
r_0	A		

i	Q	Decision
0	$\{a_0, a_1\}$	Maybe A
0	$\{a_0, a_2\}$	Maybe A
0	$\{a_1, a_2\}$	Maybe A

(b) State after receiving $\langle$P2B, 0, $A\rangle$ from a_0.

	a_0	a_1	a_2
r_0	A	A	

i	Q	Decision
0	$\{a_0, a_1\}$	Decided A
0	$\{a_0, a_2\}$	Maybe A
0	$\{a_1, a_2\}$	Maybe A

(c) State after receiving $\langle$P2B, 0, $A\rangle$ from a_1.

Figure 7. Sample proposer state tables (left) and decision tables (right) for proposer p_0 during the execution in Figure 6.

	a_0	a_1	a_2
r_0			

i	Q	Decision
0	$\{a_0, a_1\}$	Any
0	$\{a_0, a_2\}$	Any
0	$\{a_1, a_2\}$	Any

(a) Initial state.

	a_0	a_1	a_2
r_0	A		

i	Q	Decision
0	$\{a_0, a_1\}$	Maybe A
0	$\{a_0, a_2\}$	Maybe A
0	$\{a_1, a_2\}$	Maybe A

(b) State after receiving $\langle$P1B, 1, $\{r_0 : A\}\rangle$ from a_0.

	a_0	a_1	a_2
r_0	A		
r_1	A		

i	Q	Decision
0	$\{a_0, a_1\}$	Maybe A
0	$\{a_0, a_2\}$	Maybe A
0	$\{a_1, a_2\}$	Maybe A
1	$\{a_0, a_1\}$	Maybe A
1	$\{a_0, a_2\}$	Maybe A
1	$\{a_1, a_2\}$	Maybe A

(c) State after receiving $\langle$P2B, 1, $A\rangle$ from a_0.

	a_0	a_1	a_2
r_0	A		
r_1	A	A	

i	Q	Decision
0	$\{a_0, a_1\}$	Maybe A
0	$\{a_0, a_2\}$	Maybe A
0	$\{a_1, a_2\}$	Maybe A
1	$\{a_0, a_1\}$	Decided A
1	$\{a_0, a_2\}$	Maybe A
1	$\{a_1, a_2\}$	Maybe A

(d) State after receiving $\langle$P2B, 1, $A\rangle$ from a_1.

Figure 8. Sample proposer state tables (left) and decision tables (right) for proposer p_1 during the execution in Figure 6.

Relaxed Paxos can be configured with any mapping of quorums to rounds. However, the choice of quorum configuration will impact the conditions under which progress can be guaranteed. Paxos requires proposers to wait for responses from a quorum of acceptors in each phase of the algorithm and typically utilizes majority quorums as it requires all quorums to intersect, regardless of the round or phase of the algorithm. Agreement from a majority of acceptors is therefore both necessary and sufficient for a proposer to complete phase one.

Typically, descriptions of Paxos require acceptors to maintain two variables: the last round promised and the last accepted proposal (consisting of a round number and value). Relaxed Paxos instead uses a set of write-once registers to store accepted values. The nil value is used to implement phase one, without the need for a secondary set of registers.

5.1 Known results

Paxos uses the same quorums system for all phases and rounds. Instead, Flexible Paxos differentiates between the quorums used for each round and which phase of Paxos the quorum is used for. Q_r^k is the set of quorums for phase k of the round i. Flexible Paxos observed that quorum intersection is

required only between the phase one quorum for round i and the phase two quorums of rounds 0 to $i-1$. More formally, $\forall i \in \mathbb{N}_0, \forall i' \in \mathbb{N}_{<i} : Q_i^1 \cap Q_{i'}^2 \neq \emptyset$.

We can observe the same result in Relaxed Paxos. A proposer can always safely proceed to phase two of round i after receiving P1B messages from at least one acceptor in each quorum from rounds 0 to $i-1$. This is because, upon receiving $\langle \text{P1B}, i, \mathcal{R} \rangle$ from acceptor a, the proposer learns the contents of registers r_0 to r_{i-1} on acceptor a as all these registers will have already been written. It is therefore the case that, once the proposer has updated its decision table, none of the quorums over rounds 0 to $i-1$ which contain a will have the decision state of ANY. It is also that case that the decision table will never contain MAYBE v and MAYBE v' for two different values $v \neq v'$ so the proposer will always be able to choose a single safe value to write.

One implication of this result is that no P1B messages are required to complete phase one for round 0. This is illustrated in Figure 7a where the proposer p_0 was safe to proceed directly to phase two from startup.

5.2 New results

Relaxed Paxos further improves over Flexible Paxos as it also allows the proposer to safely proceed to phase two before hearing from a phase one quorums of acceptors.

We observe that it is possible for a proposer to proceed to phase two of round i without receiving a P1B messages from at least one acceptor in each quorum from rounds 0 to $i-1$. If a proposer learns that a register r_i contains a (non-nil) value v then it also learns that if any quorums from rounds 0 to i reach a decision then v must be chosen. By updating their decision table, we observe that it is no longer necessary for the proposer in phase one to intersect with the phase two quorums of registers up to r_i (inclusive). This is illustrated in Figure 8b where the proposer could safely proceed to phase two after one P1B message as the proposer reads a non-nil value from the predecessor round.

We also observe that the value selection rule (the condition that governs which values a proposer can safely write in phase two) is weaker in Relaxed Paxos than in the original Paxos protocol. Paxos permits a proposer to propose its input value in phase two only if it did not learn of any values in phase one. Relaxed Paxos also allows a proposer to propose its input value (or more generally, any value) if it knows that all values it has already learned cannot have been decided. This could be because another acceptor in each possible quorum has written nil to its register for that round.

6 Conclusion

Paxos has long been the *de facto* approach to reaching consensus, however, it is also notoriously difficult to understand. This work is the latest addition in a long line of papers that seek to explain Paxos in simpler terms, however, we are the first to extend Flexible Paxos and further relax the quorum intersection requirements of Paxos. We have re-framed the problem of consensus in terms of write-once registers and presented an abstract solution to distributed consensus comprised of four rules which an algorithm must abide by in-order to correctly solve consensus. Utilizing this abstraction, we have presented Relaxed Paxos, which further weakens Paxos's requirements for quorum intersection.

Further exploration of this result and our abstraction solution to consensus can be found in the author's thesis [9] and the extended version of the paper on arxiv [11] respectively.

7 Acknowledgements

We would like to thank Jon Crowcroft, Stephen Dolan and Martin Kleppmann for their valuable feedback on earlier iterations of this paper. This work was funded in part by EPSRC EP/R03351X/1 and EP/T022493/1.

References

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

[2] 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. URL: http://dl.acm.org/citation.cfm?id=1298455.1298487.

[3] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM. URL: http://doi.acm.org/10.1145/1281100.1281103, doi:10.1145/1281100.1281103.

[4] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3), Aug 2013. doi:10.1145/2491245.

[5] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988. doi:10.1145/42282.42283.

[6] 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, 1985.

[7] Rachid Guerraoui and Michel Raynal. The alpha of indulgent consensus. *Comput. J.*, 50(1):53–67, January 2007. URL: http://dx.doi.org/10.1093/comjnl/bxl046, doi:10.1093/comjnl/bxl046.

[8] Pat Helland. Immutability changes everything. *Queue*, 13(9):40:101–40:125, November 2015. URL: http://doi.acm.org/10.1145/2857274.2884038, doi:10.1145/2857274.2884038.

[9] Heidi Howard. *Distributed consensus revised.* PhD thesis, University of Cambridge, 2019. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-935.pdf.

[10] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible paxos: Quorum intersection revisited. In *20th International Conference on Principles of Distributed Systems (OPODIS 2016)*, volume 70, pages

25:1–25:14, Dagstuhl, Germany, 2017. URL: http://drops.dagstuhl.de/opus/volltexte/2017/7094, doi:10.4230/LIPIcs.OPODIS.2016.25.

[11] Heidi Howard and Richard Mortier. A generalised solution to distributed consensus. *CoRR*, abs/1902.06776, 2019. URL: http://arxiv.org/abs/1902.06776, arXiv:1902.06776.

[12] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998. URL: http://doi.acm.org/10.1145/279227.279229, doi:10.1145/279227.279229.

[13] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)*, 2001.

[14] Butler Lampson. The abcd's of paxos. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*, PODC '01, pages 13–, New York, NY, USA, 2001. ACM. URL: http://doi.acm.org/10.1145/383962.383969, doi:10.1145/383962.383969.

[15] 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. URL: http://dl.acm.org/citation.cfm?id=645953.675640.

[16] Hein Meling and Leander Jehl. Tutorial summary: Paxos explained from scratch. In *Proceedings of the 17th International Conference on Principles of Distributed Systems - Volume 8304*, OPODIS 2013, pages 1–10, Berlin, Heidelberg, 2013. Springer-Verlag. doi:10.1007/978-3-319-03850-6_1.

[17] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proc. USENIX Annual Technical Conference*, pages 305–320, 2014.

[18] 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. URL: http://dl.acm.org/citation.cfm?id=645954.675657.

[19] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990. URL: http://doi.acm.org/10.1145/98163.98167, doi:10.1145/98163.98167.

[20] Marc Shapiro, Nuno Preguiça, Carlos Baquero Moreno, and Marek Zawirsky. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems*, page 386–400, Berlin, Heidelberg, July 2011. Springer Berlin Heidelberg.

[21] Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Comput. Surv.*, 47(3):42:1–42:36, February 2015. URL: http://doi.acm.org/10.1145/2673577, doi:10.1145/2673577.

A Safety of the four rules for correctness

Figure 3 proposes four rules which we claim are sufficient to satisfy the safety (non-triviality and agreement) requirements of distributed consensus. We now prove each requirement in turn. We will use $a[i] = v$ to denote that register r_i on acceptor a contains the value v and $a[i] = *$ to denote that register r_i on acceptor a is unwritten.

Lemma A.1 (Satisfying non-triviality)**.** *If a value v is the output of a proposer p then v was the input of some proposer p'.*

Proof. Assume that the (non-nil) value v was the output of proposer p. According to Rule 1, at least one register must contain v. Consider the invariant that all (non-nil) registers contain proposer input values. Initially, all registers are unwritten thus this invariant holds. According to Rule 2, each proposer will only write either their input value or a value copied from another register, thus the invariant will be preserved. □

Lemma A.2 (Satisfying agreement)**.** *If two proposers, p and p', output values, v and v' (respectively), then $v = v'$.*

Proof. Assume that the (non-nil) value v was the output of proposer p and that the (non-nil) value v' was the output of proposer p'. According to Rule 1, it must be the case that $\exists i \in \mathbb{N}_0, \exists Q \in \mathcal{Q}_i, \forall a \in Q : a[i] = v$ and $\exists i' \in \mathbb{N}_0, \exists Q' \in \mathcal{Q}_i, \forall a' \in Q' : a'[i'] = v'$. Since rounds are totally ordered, it must be the case that either $i = i'$, $i < i'$ or $i > i'$. We now consider each case in turn:

Case $i = i'$:
According to Rule 3, a proposer will only write v to round i after ensuring no quorum in round i can reach a different decision. Thus $v = v'$.

Case $i < i'$:
According to Rule 4, a proposer will only write v' to round i' after ensuring no quorum in round i can reach a different decision. Thus $v = v'$.

Case $i > i'$:
This is the same as case $i < i'$ with i and i' swapped. Thus $v = v'$.

□

B Safety of decision tables rules

We have shown that the four rules for correctness are sufficient to satisfy the safety (non-triviality and agreement) requirements of consensus. We will now show that the proposer decision table rules (Fig. 5) implement the four rules for correctness (Fig. 3) and thus satisfies the safety requirements of consensus.

Lemma B.1 (Satisfying Rule 1)**.** *If the value v is the output of proposer p then p has read v from a register r_i on a quorum of acceptors $Q \in \mathcal{Q}_i$.*

Proof. Assume the value v is the output of the proposer p. There must exist a round i and quorum $Q \in \mathcal{Q}_i$ in the decision table of p with the status Decided v (Fig. 5). A quorum Q can only reach decision state Decided v if $\forall a \in Q : a[i] = v$. □

Lemma B.2 (Satisfying Rule 2)**.** *If the value v is written by a proposer p then either v is p's input value or v has been read from some register.*

Proof. Assume the value v has been written by proposer p. According to Figure 5, v must be either the input value of p or read from some register. □

Lemma B.3 (Satisfying Rule 3)**.** *If the values v and v' are decided in round i then $v = v'$.*

Proof. Assume the values v and v' are decided in round i and therefore there must be at least one acceptor a where $a[i] = v$ and one acceptor a' where $a[i] = v'$. At most one proposer

is assigned round i and the proposer will only write one (non-nil) value to i (Fig. 5) and so round i will only contain one (non-nil) value thus $v = v'$. □

Lemma B.4 (Satisfying Rule 4)**.** *If the value v is decided in round i and the (non-nil) value v' is written to round i' where $i < i'$ then $i = i'$*

We will prove this by induction over the writes to rounds $> i$. Note that this proof assumes only a partial order over writes as there may be concurrent writes to different acceptors.

Lemma B.5 (Satisfying Rule 4 - Base case)**.** *If the value v is decided in round i then the first (non-nil) value to be written to a round i' where $i < i'$ is v.*

Proof. Assume the value v is decided in round i by quorum $Q \in \boldsymbol{Q}_i$ thus at some time $\forall a \in Q : a[i] = v$. Since registers are write-once, it will always be that case that $\forall a \in Q : a[i] = v \vee *$.

Assume the value v' is written to round i' by proposer p where $i < i'$. Assume that i' is the first non-nil value to be written to any register $> i$ thus p cannot read any (non-nil) values from registers $> i$ before writing v'. We will show that $v = v'$.

Consider the decision table of proposer p when it is writing v' to i'. Since $i < i'$, the decision state of Q must be either None, Maybe v' or Decided v' (Fig. 5). We now consider each case in turn.

Case Decided v':
This decision state requires that $\forall a \in Q : a[i] = v'$. Since we know that $\forall a \in Q : a[i] = v \vee *$ then this case can only occur if $v = v'$.

Case Maybe v':
This decision state can be reached in one of two ways:
Case p read v' from register i of some acceptor a: Since at most value is written to each round, this case can only occur if $v = v'$.
Case p read v' from a register $> i$: Since v' is the first value to be written to a register $> i$, this case cannot occur.

Case None:
This decision state can be reached in one of two ways:
Case p read *nil* from register i of some acceptor $a \in Q$: Since $\forall a \in Q : a[i] = v \vee *$, this case cannot occur.
Case p read two different non-nil values from rounds $\geq i$:
Since the proposer can only read v from round i and cannot have read any non-nil values from registers sets $> i$, this case cannot occur.

Each case either requires that $v = v'$ or cannot occur, therefore it must be case that $v = v'$ □

Lemma B.6 (Satisfying Rule 4 - Inductive case)**.** *If the value v is decided in round i and all (non-nil) values written to registers $> i$ are v then the next (non-nil) value to be written to a round i' where $i < i'$ is also v.*

Since the following proof overlaps significantly with the previous proof, we have underlined the parts which have been altered.

Proof. Assume the value v is decided in round i by quorum $Q \in \boldsymbol{Q}_i$ thus at some time $\forall a \in Q : a[i] = v$. Since registers are write-once, it will always be that case that $\forall a \in Q : a[i] = v \vee *$.

Assume the value v' is written to round i' by proposer p where $i < i'$. Assume that all (non-nil) values written to registers $> i$ are v thus p can only read v from (non-nil) registers $> i$. We will show that $v = v'$.

Consider the decision table of proposer p when it is writing v' to i'. Since $i < i'$, the decision state of Q must be either None, Maybe v' or Decided v' (Fig. 5). We now consider each case in turn.

Case Decided v':
This decision state requires that $\forall a \in Q : a[i] = v'$. Since we know that $\forall a \in Q : a[i] = v \vee *$ then this case can only occur if $v = v'$.

Case Maybe v':
This decision state can be reached in one of two ways:
Case p read v' from register i of some acceptor a: Since at most value is written to each round, this case can only occur if $v = v'$.
Case p read v' from a register $> i$: Since v is the only (non-nil) value to be written to registers $> i$, then this case can only occur if $v = v'$.

Case None:
This decision state can be reached in one of two ways:
Case p read *nil* from register i of some acceptor $a \in Q$: Since $\forall a \in Q : a[i] = v \vee *$, this case cannot occur.
Case p read two different non-nil values from rounds $\geq i$:
Since the proposer can only read v from round $\geq i$, this case cannot occur.

Each case either requires that $v = v'$ or cannot occur, therefore it must be case that $v = v'$ □