

Flexible Paxos: Quorum Intersection Revisited

Heidi Howard¹, Dahlia Malkhi², and Alexander Spiegelman³

- 1 VMware Research, Palo Alto, CA, USA; and
University of Cambridge Computing Laboratory, Cambridge, UK
heidi.howard@cl.cam.ac.uk
- 2 VMware Research, Palo Alto, CA, USA
dahliamalkhi@gmail.com
- 3 VMware Research, Palo Alto, CA, USA; and
Viterbi Dept. of Electrical Engineering, Technion Haifa, Haifa, Israel
sashas@tx.technion.ac.il

Abstract

Distributed consensus is integral to modern distributed systems. The widely adopted Paxos algorithm uses two phases, each requiring majority agreement, to reliably reach consensus. In this paper, we demonstrate that Paxos, which lies at the foundation of many production systems, is conservative. Specifically, we observe that each of the phases of Paxos may use non-intersecting quorums. Majority quorums are not necessary as intersection is required only across phases.

Using this weakening of the requirements made in the original formulation, we propose Flexible Paxos, which generalizes over the Paxos algorithm to provide flexible quorums. We show that Flexible Paxos is safe, efficient and easy to utilize in existing distributed systems. We discuss far reaching implications of this result. For example, improved availability results from reducing the size of second phase quorums by one when the system size is even, while keeping majority quorums in the first phase. Another example is improved throughput of replication by using much smaller phase 2 quorums, while increasing the leader election (phase 1) quorums. Finally, non intersecting quorums in either first or second phases may enhance the efficiency of both.

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases Paxos, Distributed Consensus, Quorums

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.25

1 Introduction

Distributed consensus is the problem of reaching agreement in the face of failures. It is a common problem in modern distributed systems and its applications range from distributed locking and atomic broadcast to strongly consistent key value stores and state machine replication [36]. Lamport's Paxos algorithm [19, 20] is one such solution to this problem and since its publication it has been widely built upon in teaching, research and practice.

At its core, Paxos uses two phases, each requires agreement from a subset of participants (known as a quorum) to proceed. The safety and liveness of Paxos is based on the guarantee that any two quorums will intersect. To satisfy this requirement, quorums are typically composed of any majority from a fixed set of participants, although other quorum schemes have been proposed.

In practice, we usually wish to reach agreement over a sequence of values, known as Multi-Paxos [20]. We use the first phase of Paxos to establish one participant as a *leader* and the second phase of Paxos to propose a series of values. To commit a value, the leader must



© Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman;
licensed under Creative Commons License CC-BY

20th International Conference on Principles of Distributed Systems (OPODIS 2016).

Editors: Panagioti Fatourou, Ernesto Jiménez, and Fernando Pedone; Article No. 25; pp. 25:1–25:14

Leibniz International Proceedings in Informatics



Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



always communicate with at least a quorum of participants and wait for them to accept the value.

In this paper, we weaken the requirement in the original protocol that all quorums intersect to require only that quorums from different phases intersect. Within each of the phases of Paxos, it is safe to use disjoint quorums and majority quorums are not necessary. We will refer to this new formulation as Flexible Paxos (FPaxos) as it allows developers the flexibility to choose quorums for the two phases, provided they meet the above requirement. FPaxos is strictly more general than Paxos and FPaxos with intersecting quorums is equivalent to Paxos.

Given that Multi-Paxos and its variants are widely deployed, such a result has wide reaching practical applications. Since the second phase of Paxos (replication) is far more common than the first phase (leader election), we can use FPaxos to reduce the size of commonly used second phase quorums. For example, in a system of 10 nodes, we can safely allow any set of only 3 nodes to participate in replication, provided that we require 8 nodes to participate when recovering from leader failure. This strategy, decreasing phase 2 quorums at the cost of increasing phase 1 quorums, is referred to in the body of the paper as *simple quorums*.

The simple quorum system reduces latency, as leaders will no longer be required to wait for a majority of participants to accept proposals. Likewise, it improves steady state throughput as disjoint sets of participants can now accept proposals, enabling better utilization of participants and decreased network load. The price we pay for this is reduced availability as the system can tolerate fewer failures whilst recovering from leader failure.

However, it is not always necessary to compromise availability for improved steady state performance. For example, in a system of 10 nodes, we can safely use any set of 5 nodes to form phase 2 quorums and any set of 6 nodes to form phase 1 quorums. This improves both the performance and availability of the second phase, without hurting availability of the first phase at all. In the paper we also illustrate that surprisingly, there are quorum systems such as grid quorums, in which FPaxos allows us to decrease the quorum sizes of both phases. Furthermore, in this quorum system, the quorums within either phase do not intersect with each other.

In the following section we outline the basic Paxos algorithm using the standard terminology. Readers who are already familiar with the algorithm should proceed directly to the next section. In §3 we describe the observation in detail and then in §4 motivate why such flexibility is useful in practice. §5 gives an informal description of why it is safe to weaken Paxos's assumption on quorum intersection. In §6 we evaluate a naïve implementation of FPaxos and demonstrate its usefulness. §7 outlines how to dynamically choose quorums and §8 relates FPaxos to the existing work in the field. A TLA+ [21] specification of the FPaxos algorithm, which has been model checked against our safety assumption, can be found in [11].

2 Paxos

We wish to decide a single value v between a set of processes. The system is asynchronous, each process may fail and the messages passed between them may be lost. Each process has one or more roles. We have three roles: the proposer, a process who wishes to have a particular value chosen, the acceptor, a process which agrees and persists decided values or the learner, a process wishing to learn the decided value.

A proposer who has a candidate value will try to propose the value to the acceptors. If a value has already been chosen, the proposer will instead learn it. The process of proposing a

value has two stages: phase 1 and phase 2, each phase requires a majority of acceptors to agree in order to proceed. We will now look at each of these stages in details:

Phase 1 – Prepare & Promise

- (i) A proposer selects a unique proposal number p and sends $prepare(p)$ to the acceptors.
- (ii) Each acceptor receives $prepare(p)$. If p is the highest proposal number promised, then p is written to persistent storage and the acceptor replies with $promise(p', v')$. (p', v') is the last accepted proposal (if present) where p' is the proposal number and v' is the corresponding proposed value.
- (iii) Once the proposer receives $promise$ from the majority of acceptors, it proceeds to phase two. Otherwise, it may try again with higher proposal number.

Phase 2 – Propose & Accept

- (i) The proposer must now select a value v . If more than one proposal was returned in phase 1 then it must choose the value associated with the highest proposal number. If no proposals were returned, then the proposer can choose its own value for v . The proposer then sends $propose(p, v)$ to the acceptors.
- (ii) Each acceptor receives a $propose(p, v)$. If p is equal to or greater than the highest promised proposal number, then the promised proposal number and accepted proposal is written to persistent storage and the acceptor replies with $accept(p)$.
- (iii) Once the proposer receives $accept(p)$ from the majority of acceptors, it learns that the value v is decided. Otherwise, it may try phase 1 again with a higher proposal number.

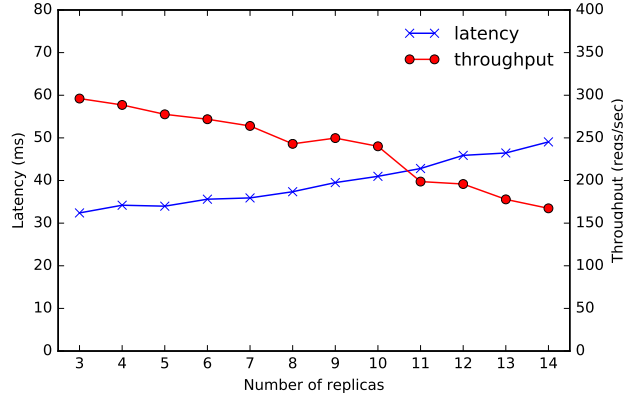
Paxos guarantees that once a value is decided, the decision is final and no different value can be chosen. Paxos will reach agreement provided that $\lfloor N/2 \rfloor + 1$ acceptors out of N acceptors are up and are able to communicate. Proving progress requires us to make some assumptions about the synchrony of the system, as we cannot guarantee progress in a truly asynchronous systems [7].

Usually, we wish to reach agreement over a sequence of values, which we will refer to as slots. We could use distinct instances of Paxos to decide each value in the sequence i.e. the i^{th} slot is decided by the i^{th} instance of Paxos. In practice however, we can do much better and this is referred to as Multi-Paxos.

The first phase of Paxos is independent of the value proposed for any given instance, therefore phase 1 can be executed prior to knowledge of which value to propose. Furthermore, we can aggregate phase 1 over a series of slots. We refer to a proposer who has completed phase 1 as a *leader*. To avoid loss of generality, we introduce another agent, the *client* who is the origin of values for proposal. Clients may be external to the system or co-located with other processes such as the proposers.

Figure 1 illustrates how Multi-Paxos performs in practice. The x-axis shows the number of replicas in the system, each replica performs the roles of proposer, acceptor and learner. The blue line indicates the commit latency observed by the client and the red line indicates the average request throughput. As we would expect, increasing the number of replicas will increase latency and decrease throughput. These findings are consistent with previous studies [30, 27].

A *quorum system* is the method by which we choose which sets of acceptors are able to form valid quorums. It has been observed that Paxos can be generalized to replace majority quorums with any quorum system which guarantees that any two quorums will have a non-empty intersection [20, 22]. The fundamental theorem of quorum intersection



■ **Figure 1** Performance of LibPaxos3 for varying system sizes. Details of the experimental setup are given in §6.

states that its resilience is inversely proportional to the load on (hence the throughput of) participants [32]. Therefore, with Paxos and its intersecting quorums, one can only hope to increase throughput by reducing the resilience, or vice versa. In the rest of this paper, we show that by weakening the quorum intersection requirement, we can break away from the inherent trade off between resilience and performance.

3 FPaxos

In this section, we observe that the usual description of Paxos (as given in §2) is more conservative than is necessary. To explain this observation, we will differentiate between the quorum used by the first phase of Paxos, which we will refer to as $Q1$ and the quorum for second phase, referred to as $Q2$.

Paxos uses majority quorums of acceptors for both $Q1$ and $Q2$. By requiring that quorums contain at least a majority of acceptors we can guarantee that there will be at least one acceptor in common between any two quorums. Paxos’s proof of safety and progress is built upon this assumption that all quorums intersect.

We observe that it is only necessary for phase 1 quorums ($Q1$) and phase 2 quorums ($Q2$) to intersect. There is no need to require that $Q1$ ’s intersect with each other nor $Q2$ ’s intersect with each other. We refer to this as Flexible Paxos (FPaxos) and it generalizes the Paxos algorithm. If we allow any set of at least $\lfloor N/2 \rfloor + 1$ acceptors to form a $Q1$ or $Q2$ quorum in FPaxos, then FPaxos is equivalent to Paxos.

Using this observation, we can make use of many non-intersecting quorum systems. In its most straight-forward application, we can simply decrease the size of $Q2$ at the cost of increasing the size of $Q1$ quorums.

As we discussed earlier, the second phase of Paxos (replication) is far more frequent than the first phase (leader election) in Multi-Paxos. Therefore, reducing the size of $Q2$ decreases latency in the common case by reducing the number of acceptors required to participate in replication, improves system tolerance to slow acceptors and allows us to use disjoint sets of acceptors for higher throughput. The price we pay for this is requiring more acceptors to participate when we need to establish a new leader. Whilst electing a new leader is a rare event in a stable system, if sufficient failures occur that we cannot form a $Q1$ quorum, then we cannot make progress until some of the acceptors recover.

Like Paxos, the system is able to make progress provided that at least enough acceptors are up and able to communicate to form both $Q1$ and $Q2$ quorums. Unlike Paxos, we are able to make progress within a given phase, provided we are able to form quorums corresponding to that phase. More concretely, if sufficient failures have occurred such that a proposer can no longer form $Q1$ quorums but is able to form the smaller $Q2$ quorums, the system can continue to safely make progress until a new leader is required. If the acceptors recover before the current leader fails, then the system suffers no loss in availability as a result.

4 Implications

We will now consider the practical implication of observing that quorums intersection is required only between the two phases of Paxos. There already exists an extensive literature on quorum systems from the fields of databases and data replication, which can now be more efficiently applied to the field of consensus. Interesting example systems include weighted voting [9], hierarchies [16] and crumbling walls [35]. For now however, we will illustrate the utility of FPaxos by considering three naïve example quorum systems: (1) majority quorums; (2) simple quorums and (3) grid quorums.

4.1 Majority quorums

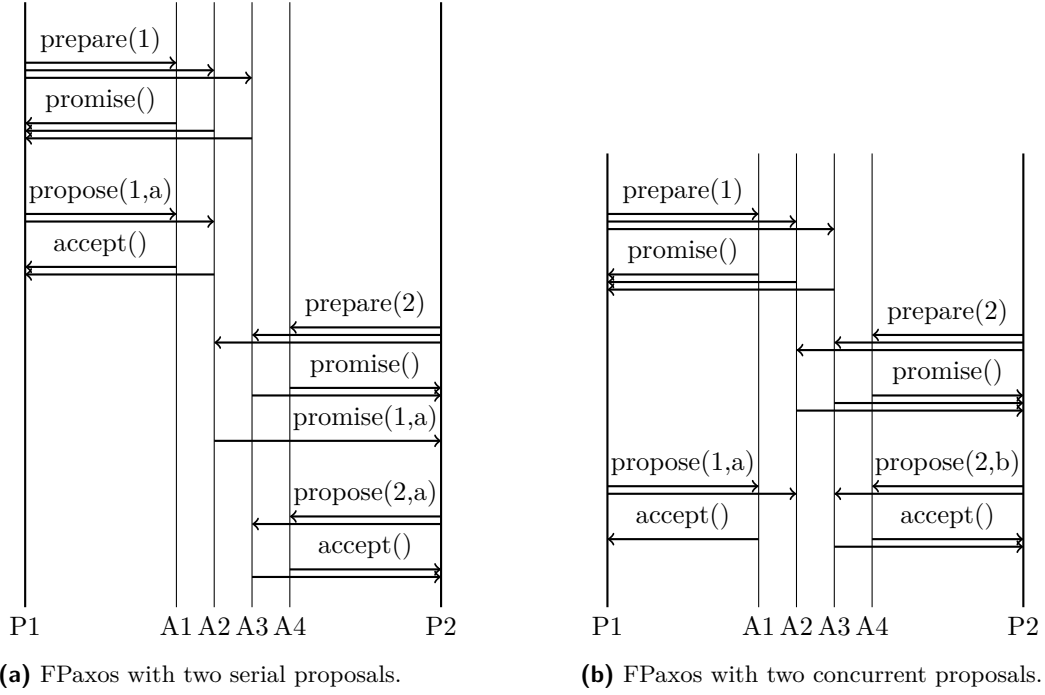
Currently, Paxos requires us to use quorums of size $N/2 + 1$ when the number of acceptors N is even¹. Using our observation, we can safely reduce the size of $Q2$ by one from $N/2 + 1$ to $N/2$ and keep $Q1$ the same. Such a change would be trivial to implement and by reducing the number of acceptors required to participate in replication, we can reduce latency and improve throughput. Furthermore, we have also improved the fault tolerance of the system. As with Paxos, if at most $N/2 - 1$ failures occur then we are guaranteed to be able to make progress. However unlike with Paxos, if exactly $N/2$ acceptors fail and the leader is still up then we are able to continue to make progress and suffer no loss of availability.

Figure 2 shows two example traces of FPaxos with majority quorums in practice. As the system is comprised of four acceptors, FPaxos uses a majority (3 acceptors) for $Q1$ but requires only two acceptors for $Q2$. In the examples, the two proposers wish to commit conflicting proposals. In Figure 2a, proposer one is first to execute FPaxos and its value a is committed. Later, proposer two executes a round of Paxos and learns the value. In Figure 2b, both proposers successfully execute the first phase of FPaxos and simultaneously submit conflicting proposed values to the disjoint sets of acceptors. Both $Q2$ s will intersect with the two $Q1$ s, so only one of them will be successful. In the given example, acceptor two will not accept *propose*(1, a) as it has already promised to *prepare*(2). The unsuccessful proposer can retry with a higher proposal number and learn the chosen value.

4.2 Simple quorums

We will use the term *simple quorums* to refer to a quorum systems where any acceptor is able to participate in a quorum and each acceptor's participation is counted equally. Simple quorums are a straightforward generalization of majority quorums. Paxos requires that all quorums intersect, and therefore, as we have previously discussed, each quorum must contain at least a strict majority of acceptors to meet this requirement.

¹ Lamport observed that majorities can be extended to include exactly half of the sets of size $N/2$ [17].



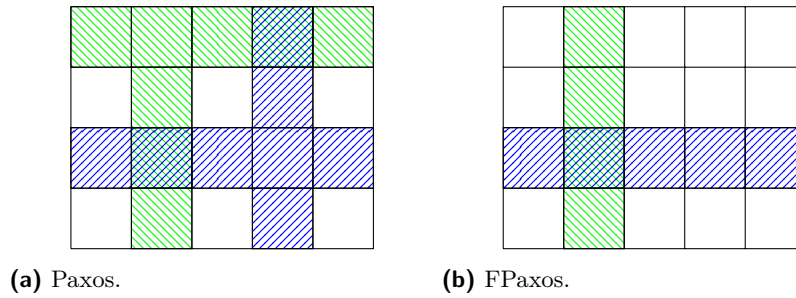
■ **Figure 2** Sample executions of FPaxos using improved majority quorums. The system is comprised of four acceptors (A1-A4) and two proposers (P1,P2).

In contrast, FPaxos requires only that quorums from different phases intersect. Therefore, FPaxos with simple quorums must require that $|Q1| + |Q2| > N$. We know that in practice the second phase is much more common than the first phase so we allow $|Q2| < N/2$ and increase the size of $Q1$ accordingly. For a given size of $Q2$ and number of acceptors N , the minimum size of our first phase quorum is $|Q1| = N - |Q2| + 1$. FPaxos will always be able to handle up to $|Q2| - 1$ failures. However, if between $|Q2|$ to $N - |Q2|$ failures occur, we can continue replication until a new leader is required.

As has been previously observed [26], we do not need to send *prepare* and *propose* messages to all acceptors, only to at least $|Q1|$ or $|Q2|$ acceptors. If any of these acceptors do not reply, then the leader can send the messages to more acceptors. This reduces the number of messages from $4 \times N$ to $(2 \times |Q1|) + (2 \times |Q2|)$. This comes at the cost of increased latency, as the leader may not choose the fastest acceptors and must retransmit when failures occur.

4.3 Grid quorums

The key limitation of simple quorums is that reducing the size of the $Q2$ requires a corresponding increase in the size of $Q1$ to continue to ensure intersection. Grid quorums are an example of an alternative quorum system. Grid quorums can reduce the size of $Q1$ by offering a different trade off between quorum sizes, flexibility when choosing quorums and failure tolerance. Grid quorum schemes arrange the N nodes into a matrix of N_1 columns by N_2 rows, where $N_1 \times N_2 = N$ and quorums are composed of rows and columns. As with many other quorum systems, grid quorums restrict which combinations of acceptors can form valid quorums. This restriction allows us to reduce the size of quorums whilst still ensuring that they intersect.



■ **Figure 3** Example of using a 5 by 4 grid to form quorums for a system of 20 acceptors.

Paxos requires that all quorums intersect thus one suitable grid scheme would require one row and one column to form a quorum². Figure 3a shows an example $Q1$ quorum and $Q2$ quorum using this scheme. This would reduce the size of a quorum from the majority of N to $N_1 + N_2 - 1$. The number of failures which could be tolerated range from $\text{MIN}(N_1, N_2)$, where one node from every row or every column fails to $(N_1 - 1) \times (N_2 - 1)$, leaving only one row and one column remaining.

In FPaxos, we can safely reduce our quorums to one row of size N_1 for $Q1$ and one column of size N_2 for $Q2$, examples are shown in Figure 3b. This construction is interesting as quorums from the same phase will never intersect, and may be useful in practice for evenly distributing the load of FPaxos across a group of acceptors. With simple quorums, a system cannot recover from leader failure whilst any set of $|Q2| = N/2$ acceptors have failed. Now with grid quorums, we are no longer treating all failures equally, it matters which of the acceptors have failed, not just how many have failed. Recall, that we are able to make progress in a given phase, provided we can still form a quorum for that phase. For example, let us consider if four acceptors in either of grids from Figure 3 were to fail. If these failures occur across two columns then both systems will make progress. If all the failed nodes are within one column then no progress will be made by Paxos but FPaxos will continue until a new leader is needed. Likewise, if all the nodes in a given row were to fail, FPaxos would be able to complete $Q1$ and thus recover all past decisions, it can then safely fall back to a reconfiguration protocol to remove or replace the failed acceptors and continue to make progress. In practice, failures are not independent and so we can distribute acceptors across the machines, racks or even data centers to minimize the likelihood of simultaneous failure.

By way of a thought experiment, let us consider setting $N_1 = 1$ and $N_2 = N$ when using grid quorums or equivalently setting $|Q1| = N$ and $|Q2| = 1$ with simple quorums. Any single acceptor will be sufficient to form a $Q2$, however every acceptor must participate in $Q1$. In practice, this would allow all acceptors to learn the decided value in a single hop, however we would be unable to recover from leader failure until every acceptor is up.

Alternatively, let us consider setting $N_1 = N$ and $N_2 = 1$ when using grid quorums or equivalently setting $|Q1| = 1$ and $|Q2| = N$ with simple quorums. This would require every acceptor to participate in $Q2$ but only a single acceptor is needed for $Q1$. If any acceptors are still up, then we can complete $Q1$ locally and learn past decisions. As it has been previously observed [26, 24], such a construction allows us to tolerate f failures with only $f + 1$ acceptors instead of $2f + 1$.

² In practice, it is sufficient to use one row plus any choice of one grid item from each row below it. The average quorum size would become $N_1 + (1/2)N_2$, although the worst case is still $N_1 + N_2 - 1$.

5 Safety

Lamport's proof of safety for Paxos does not use the full strength of the assumptions made, namely that all quorums will intersect. For the sake of completeness, in this section we outline the proof of safety for FPaxos.

For FPaxos to be safe, every decision that is reached must be final. In other words, once a value has been decided, no different value can be decided. This can be formally expressed as the following requirement:

► **Theorem 1.** *If value v is decided with proposal number p and v' is decided with proposal number p' then $v = v'$*

For a given value v to be decided, it must first have been proposed. Thus the following requirement is strictly stronger:

► **Theorem 2.** *If value v is decided with proposal number p then for any message $\text{propose}(p', v')$ where $p' > p$ then $v = v'$*

Proof is by contradiction, that is, assume $v \neq v'$. We will consider the smallest proposal number $p' > p$ for which such a message is sent.

Let $\mathcal{Q}_1$ and $\mathcal{Q}_2$ be the sets of all valid phase 1 and phase 2 quorums respectively and $\mathcal{A}$ be the set of acceptors. Quorums are valid provided that:

$$\forall Q_1 \in \mathcal{Q}_1 : Q_1 \subseteq \mathcal{A} \quad (1)$$

$$\forall Q_2 \in \mathcal{Q}_2 : Q_2 \subseteq \mathcal{A} \quad (2)$$

$$\forall Q_1 \in \mathcal{Q}_1, \forall Q_2 \in \mathcal{Q}_2 : Q_1 \cap Q_2 \neq \emptyset \quad (3)$$

Equation 1 specifies that every possible phase 1 quorum is a subset of the acceptors, likewise for equation 2. Equation 3 specifies that all possible combinations consisting of a phase 1 and a phase 2 quorum will intersect in at least one acceptor.

Let $Q_{p,2}$ be the phase 2 quorum used by proposal number p and $Q_{p',1}$ be the phase 1 quorum used by proposal number p' . Let $\bar{A}$ be the set of acceptors which participated both in the phase 2 quorum used by proposal number p and phase 1 quorum used by proposal number p' , thus $\bar{A} = Q_{p,2} \cap Q_{p',1}$. Since $Q_{p,2} \in \mathcal{Q}_2$ and $Q_{p',1} \in \mathcal{Q}_1$ then we can use equation 3 to infer that at least one acceptor must participate in both quorums, $\bar{A} \neq \emptyset$.

Let us consider the ordering of events from the perspective of one acceptor acc where $acc \in \bar{A}$. It is either the case that they receive $\text{prepare}(p')$ first or $\text{propose}(p, v)$ first. We will consider each of these cases separately:

Case 1: Acceptor acc receives $\text{prepare}(p')$ before it receives $\text{propose}(p, v)$. When acc receives $\text{propose}(p, v)$, its last promised proposal will be p' or higher. As $p' > p$ then it will not accept the proposal from p , however as $acc \in Q_{p,2}$ it must accept $\text{propose}(p, v)$. This is a contradiction thus it cannot be the case.

Case 2: Acceptor acc receives $\text{propose}(p, v)$ before it receives $\text{prepare}(p')$. When acc receives $\text{prepare}(p')$, there are two cases. Either:

Case 2a: The last promised proposal by acceptor acc is already higher than p' . Then it will not accept the prepare from p' , however as $acc \in Q_{p',1}$ it must accept $\text{prepare}(p')$. This is a contradiction thus it cannot be the case.

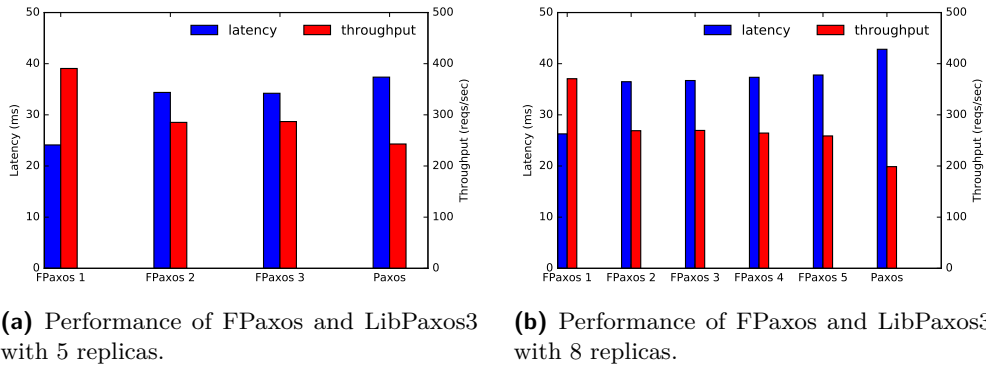


Figure 4 Throughput and average latency of FPaxos with various simple quorum sizes and LibPaxos3.

Case 2b: The last promised proposal by acceptor acc is less than p' then it will reply with $promise(q, v)$ where $p \leq q < p'$. The value v will be the same the one acc accepted with p , under the minimality hypothesis on p' .

$acc \in Q_{p',1}$ therefore $promise(q, v)$ will be at least one of the responses received by the proposer of p' . If this is the only accepted value returned, then its value v will be chosen. Other proposals may also be received for members of $Q_{p',1}$. Recall that $p < p'$. For each other proposal (q', v'') received, either:

Case (i) $q' < q$: These proposal will be ignored as the proposer must choose the value associated with the highest proposal.

Case (ii) $p' < q'$: This case cannot occur as an acceptor will only reply to $prepare(p')$ when last promised is $< p'$.

Case (iii) $p < q' < p'$: For an acceptor to have accepted (q', v'') then it must have first been proposed. This is impossible by the minimality assumption on p' .

Thus the value v will be chosen, in contradiction to the assumption that $propose(p', v')$ was sent.

We model checked the formal TLA+ [21] specification of the single-valued FPaxos protocol with disjoint quorums and the requirement 2 was preserved. The FPaxos TLA+ specification, given in [11], is only a minor adaptation of the Paxos specification, given in [21].

6 Prototype

We implemented a naïve FPaxos by modifying LibPaxos3³, a commonly benchmarked Multi-Paxos implementation. Our modification simply generalized over the size of $Q1$ and $Q2$. The simple quorums were naïvely chosen at random and messages were sent only to a quorum of nodes.

LibPaxos3 is Multi-Paxos implementation in C which uses TCP/IP for transport. For each experiment, we tested N replicas, where each replica is an acceptor, a learner and a

³ LibPaxos3 source code <https://bitbucket.org/sciascid/libpaxos>.

possible proposer. We used request sizes of 64 bytes with 10 requests in progress at any given time and all request were first sent to the same replica. Our experiments were ran within a single linux VM with a single core and 1GB of RAM, we used mininet ⁴ to simulate a 10 Mbps network with 20 ms round trip time. Each test was run for 120 seconds, we discard the first and last 10 second to measure the system during its steady state.

Figure 4 show the steady state performance of Paxos and FPaxos with varying sizes of simple quorums for $Q2$. These results are as we would expect: by reducing the size of the $Q2$ quorum, we send fewer messages and thus increase throughput and decrease latency.

It is worth noting that this is not the complete picture. First, FPaxos outperforms vanilla LibPaxos even with identical quorum sizes, because FPaxos sends messages only to a quorum of replicas unlike LibPaxos3 which sends messages to all replicas. When utilizing this optimization in practice, one may need to carefully trade the strategy for finding quorums in realistic settings, and consider replica failure, relative replica speeds and communication delays. Second, unlike Paxos, FPaxos with $Q2$ of size 2 would not be able to elect a new leader when two acceptors have failed. On the other hand, in a system of 8 replicas, FPaxos with $Q2$ of size 4 handles more failures than Paxos, decreases latency (from 42ms to 37 ms) and increases throughput (from 198 to 264 reqs/sec).

This prototype demonstrates that implementing a naïve FPaxos is trivial. We show that even a very naïve implementation improves performance and we believe that systems designed for FPaxos will see far greater performance, particularly by taking advantage of using disjoint set of acceptors and smarter quorum construction techniques to improve failure tolerance. Our prototype source code and associated materials are available online⁵.

7 Enhancements

We observe that the safety of FPaxos relies only on the assumption that a given $Q1$ will intersect with all $Q2$ s with lower proposal numbers. Therefore, we could further weaken the quorum requirements if a proposer was able to learn which $Q2$ s have been used with smaller proposal numbers. We would then require only that a proposer's $Q1$ intersect with these instead of all possible $Q2$ s.

In order to take advantage of this, we can enhance FPaxos with a mechanism for leaders to select quorum(s) and to announce their selection. There are many ways this could be implemented, but for safety the mechanism for a leader to make its quorum selection known must be weaved carefully into the leader election protocol. Details are left out of the scope of this paper. Briefly, it would be akin to Paxos reconfiguration and achieved by adopting the principles of Vertical Paxos [24].

The implications of this enhancement can be far reaching. For example, in a system of $N = 100f$ nodes, a leader may start by announcing a fixed $Q2$ of size $f + 1$ and all higher proposal numbers (and readers) will need to intersect with only this $Q2$. This allow us to tolerate $N - f$ failures. Likewise, a leader may choose a small set of $Q2$'s and announce all of them, allowing more flexibility in phase 2 at the cost of less availability in phase 1. A leader may also change its quorum selection over time using the dynamic selection mechanism.

We expect that these enhancements and others may open many new possibilities for practical system designs in the future.

⁴ <http://mininet.org>

⁵ <https://github.com/fpaxos>

8 Related Works

The insightful State Machine Replication (SMR) paradigm [18, 37] underlies many reliable systems, including pioneering works in distributed systems field like Viewstamped Replication [33] and Isis [3]. The Paxos algorithm provides the algorithmic solution for many production systems architected as replicated state machines. SMR must solve a core ingredient, agreement, which Dwork et al. [6] solved under minimal synchrony assumptions, and which is the basis for the single position agreement protocol (called Synod) in Paxos [19]. In the decades following its invention, the Paxos algorithm has been extensively researched: it has been explained in simpler terms [20, 41], optimized for practical systems [4, 12, 14, 34] and extended to handle reconfiguration [24] and arbitrary failures [5].

Many variants of Paxos were proposed. Cheap Paxos [26] fixes a single phase 2 quorum until a leader replacement occurs. Fast Paxos [22] has a leaderless fast-path protocol which utilizes fast-track phase 2 quorums of size $f + \lceil \frac{f+1}{2} \rceil$. Mencius [27] uses a revolving leader regime. Ring-Paxos [29, 28] applies the idea in Cheap Paxos [26] to a ring overlay using network-level multicast. Chain Replication [43] daisy-chains acceptors and collapses the two phases into one chain sweep. Generalized Paxos [23] extends state-machine replication with commutative commands. Egalitarian Paxos [31] extends Generalized Paxos with fast-track quorums whose size is $f + \lfloor \frac{f+1}{2} \rfloor$, although this can be further reduced [40]. EVE [15] optimistically concurrently agrees on commands and later resolves conflicts in case they do not commute. Corfu [2] lets the leader delegate its exclusive authority to any proposer in order to yield better parallelism. There are many other variants; a comprehensive taxonomy of Paxos variants is given in [42]. These previous works were built on the foundations presented in the pioneering protocols [33, 3, 6, 19], and focused on enhancing them in order to achieve better performance. Our new observation revisited the foundations and generalized them; it is completely orthogonal and can be integrated into previous protocols as well as to real production systems in order to further improve performance.

The SMR reconfiguration problem was addressed in several previous works. Some use consensus commands to agree on next configurations [33, 25, 24], whereas others use the first phase to determine which quorum (out of a fix set of quorums) will be used in the second phase [26, 29]. A general framework for reconfiguration that separates the steady state agreement mechanism from the reconfiguration event appears in [24]. Reconfiguration for other fault tolerant services was also previously investigated, e.g., in [10, 1, 13, 39, 8]. As discussed in §7, the ideas in these works can be adopted in order to enhance FPaxos into a reconfigurable and dynamic system.

To the best of our knowledge, we are the first to prove and implement this generalization of Paxos. During the preparation of this publication, Sougoumarane independently made the same observation on which this work is based and released a blog post summarizing [38] it for the systems community.

9 Conclusion

In this paper we have described FPaxos, a generalization of the widely adopted Paxos algorithm, which no longer requires that quorums from the same Paxos phase intersect. We believe this result has wide ranging consequences.

Firstly, over the last two decades Multi-Paxos has been widely studied, deployed and extended. Generalizing existing systems to use FPaxos should be quite straightforward. Exposing replication (phase 2) quorum size to developers would allow them to choose their own trade off between failure tolerance and steady state latency.

Secondly, by no longer requiring replication quorums to intersect, we have removed an important limit on scalability. Through smart quorum construction and pragmatic system design, we believe a new breed of scalable, resilient and performant consensus algorithms is now possible.

Acknowledgements. We wish to thank the following people for their feedback: Jean Bacon, Jon Crowcroft, Stephen Dolan, Matthew Grosvenor, Anil Madhavapeddy, Sugu Sougoumarane and Igor Zablotchi.

References

- 1 Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. *J. ACM*, 58(2):7:1–7:32, April 2011. doi:10.1145/1944345.1944348.
- 2 Mahesh Balakrishnan, Dahlia Malkhi, John D. Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. Corfu: A distributed shared log. *ACM Trans. Comput. Syst.*, 31(4):10:1–10:24, December 2013.
- 3 Ken Birman and Thomas Joseph. *Exploiting virtual synchrony in distributed systems*, volume 21. ACM, 1987.
- 4 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>.
- 5 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI’99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association. URL: <http://dl.acm.org/citation.cfm?id=296806.296824>.
- 6 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.
- 7 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.
- 8 Eli Gafni and Dahlia Malkhi. Elastic configuration maintenance via a parsimonious speculating snapshot solution. In *Proceedings of the 29th International Symposium on Distributed Computing*, pages 140–153. Springer, 2015.
- 9 David K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, SOSP’79, pages 150–162, New York, NY, USA, 1979. ACM. doi:10.1145/800215.806583.
- 10 Seth Gilbert, Nancy A. Lynch, and Alexander A. Shvartsman. RAMBO: A robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing*, 23(4):225–272, 2010.
- 11 Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible paxos: tlplus specification. URL: <https://github.com/fpaxos/fpaxos-tlplus>.
- 12 Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC’10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association. URL: <http://dl.acm.org/citation.cfm?id=1855840.1855851>.
- 13 Leander Jehl, Roman Vitenberg, and Hein Meling. Smartmerge: A new approach to reconfiguration for atomic storage. In *International Symposium on Distributed Computing*, pages 154–169. Springer, 2015.

- 14 Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 245–256. IEEE, 2011.
- 15 Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All about eve: execute-verify replication for multi-core servers. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 237–250, 2012.
- 16 Akhil Kumar. Hierarchical quorum consensus: A new algorithm for managing replicated data. *IEEE Trans. Comput.*, 40(9):996–1004, September 1991. doi:10.1109/12.83661.
- 17 Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks (1976)*, 2(2):95–114, 1978. doi:10.1016/0376-5075(78)90045-4.
- 18 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.
- 19 Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998. doi:10.1145/279227.279229.
- 20 Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)*, 2001.
- 21 Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- 22 Leslie Lamport. Fast paxos. Technical Report MSR-TR-2005-112, Microsoft Research, 2005.
- 23 Leslie Lamport. Generalized consensus and paxos. Technical Report MSR-TR-2005-33, Microsoft Research, March 2005.
- 24 Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, PODC’09, pages 312–313, New York, NY, USA, 2009. ACM. doi:10.1145/1582716.1582783.
- 25 Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a state machine. *ACM SIGACT News*, 41(1):63–73, 2010.
- 26 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. URL: <http://dl.acm.org/citation.cfm?id=1009382.1009745>.
- 27 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. URL: <http://dl.acm.org/citation.cfm?id=1855741.1855767>.
- 28 Parisa Jalili Marandi, Marco Primi, and Fernando Pedone. Multi-ring paxos. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12. IEEE, 2012.
- 29 Parisa Jalili Marandi, Marco Primi, Nicolas Schiper, and Fernando Pedone. Ring paxos: A high-throughput atomic broadcast protocol. In *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, pages 527–536. IEEE, 2010.
- 30 P.J. Marandi, M. Primi, N. Schiper, and F. Pedone. Ring paxos: A high-throughput atomic broadcast protocol. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 527–536, June 2010. doi:10.1109/DSN.2010.5544272.
- 31 Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Op-*

- erating Systems Principles*, SOSP'13, pages 358–372, New York, NY, USA, 2013. ACM. doi:10.1145/2517349.2517350.
- 32 Moni Naor and Avishai Wool. The load, capacity, and availability of quorum systems. *SIAM J. Comput.*, 27(2):423–447, April 1998. doi:10.1137/S0097539795281232.
 - 33 Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 8–17. ACM, 1988.
 - 34 Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proc. USENIX Annual Technical Conference*, pages 305–320, 2014.
 - 35 David Peleg and Avishai Wool. Crumbling walls: A class of practical and efficient quorum systems. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC'95, pages 120–129, New York, NY, USA, 1995. ACM. doi:10.1145/224964.224978.
 - 36 Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990. doi:10.1145/98163.98167.
 - 37 Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.
 - 38 Sugu Sougoumarane. A more flexible paxos. [Online; accessed 13-Aug-2016]. URL: <http://ssougou.blogspot.com/2016/08/a-more-flexible-paxos.html>.
 - 39 Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. Dynamic reconfiguration: A tutorial. *OPODIS 2015*, 2015.
 - 40 P. Sutra and M. Shapiro. Fast genuine generalized consensus. In *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on*, pages 255–264, Oct 2011. doi:10.1109/SRDS.2011.38.
 - 41 Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Comput. Surv.*, 47(3):42:1–42:36, February 2015. doi:10.1145/2673577.
 - 42 Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Comput. Surv.*, 47(3):42:1–42:36, February 2015. doi:10.1145/2673577.
 - 43 Robbert Van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, volume 4, pages 91–104, 2004.