Matchmaker Paxos: A Reconfigurable Consensus Protocol

Michael Whittaker[1], Neil Giridharan[1], Adriana Szekeres[2], Joseph M. Hellerstein[1], Heidi Howard[3], Faisal Nawab[4], Ion Stoica[1]

[1] University of California, Berkeley, [2] University of Washington, [3] University of Cambridge, [4] University of California, Santa Cruz

ABSTRACT

State machine replication protocols, like MultiPaxos and Raft, are at the heart of nearly every strongly consistent distributed database. To tolerate machine failures, these protocols must replace failed machines with live machines, a process known as reconfiguration. Reconfiguration has become increasingly important over time as the need for frequent reconfiguration has grown. Despite this, reconfiguration has largely been neglected in the literature. In this paper, we present Matchmaker Paxos and Matchmaker MultiPaxos, a reconfigurable consensus and state machine replication protocol respectively. Our protocols can perform a reconfiguration with little to no impact on the latency or throughput of command processing; they can perform a reconfiguration in one round trip (theoretically) and a few milliseconds (empirically); they provide a number of theoretical insights; and they present a framework that can be generalized to other replication protocols in a way that previous reconfiguration techniques can not. We provide proofs of correctness for the protocols and optimizations, and present empirical results from an open source implementation.

PVLDB Reference Format:
Michael Whittaker, Neil Giridharan, Adriana Szekeres, Joseph M. Hellerstein, Heidi Howard, Faisal Nawab, and Ion Stoica. Matchmaker Paxos: A Reconfigurable Consensus Protocol. *PVLDB*, 13(xxx): xxxx-yyyy, 2020.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

1. INTRODUCTION

Strongly consistent distributed databases—like Spanner [6], CockroachDB [11], and FoundationDB [5]—are becoming increasingly adopted in industry and increasingly studied in academia. All of these distributed databases rely on some state machine replication protocol, like MultiPaxos [12] or Raft [28], to keep multiple replicas of their data in sync. To ensure system availability, these replication protocols must be able to dynamically replace failed machines with live machines, a process known as reconfiguration.

Reconfiguration is an old problem, but recent computing trends make it a particularly timely topic of research. Historically, state machine replication protocols were deployed on a fixed set of machines, and reconfiguration was used only to replace failed nodes with live nodes – an infrequent occurrence. Recently however, systems have become increasingly elastic, and the need for frequent reconfiguration has grown. These elastic systems don't just perform reconfigurations *reactively* when machines fail; they reconfigure *proactively*. For example, cloud databases can proactively request more resources to handle workload spikes, and orchestration tools like Kubernetes [10] are making it easier to build these types of elastic systems. Similarly, in environments with short-lived cloud instances—as with serverless computing and spot instances—and in mobile edge and Internet of Things settings, protocols must adapt to a changing set of machines much more frequently.

Despite the (increasing) importance of reconfiguration, it has largely been neglected by current academic literature. Researchers have invented dozens of state machine replication protocols, yet many papers either discuss reconfiguration briefly with no evaluation [24, 30, 32, 31], propose theoretically safe but inefficient reconfiguration protocols [14, 19], or do not discuss reconfiguration at all [15, 21, 22, 2, 1]. In recent years, state-of-the-art databases—like MDCC [9], Janus [25], and TAPIR [38]—have begun to adopt more sophisticated replication protocols, but these are exactly the protocols for which there are not well-established, high performance techniques for reconfiguration. Inventing new reconfiguration protocols for these new databases is challenging. Reconfiguration protocols are notoriously complicated and hard to get right. In one recent anecdote, Raft's authors proposed a "simpler single-server" reconfiguration protocol to replace their original, more complicated protocol, but this simpler protocol turned out to be unsafe [27].

In this paper, we present Matchmaker Paxos and Matchmaker MultiPaxos, a reconfigurable consensus protocol and state machine replication protocol respectively. Our protocols have the following desirable properties:

- **Little to No Performance Degradation.** Matchmaker MultiPaxos can perform a reconfiguration without significantly degrading the throughput or latency of processing client commands. For example, we show that reconfiguration has less than a 2% effect on median and standard deviation latency measurements (Section 8).

- **Quick Reconfiguration.** Matchmaker MultiPaxos can perform a reconfiguration quickly. Theoretically, reconfiguring to a new set of machines takes one round trip of communication (Section 4). Empirically, this requires only a few milliseconds within a single data center (Section 8).

Proceedings of the VLDB Endowment, Vol. 13, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

- **Theoretical Insights.** Matchmaker Paxos offers theoretical insights into many existing protocols (Section 7). It significantly extends Vertical Paxos [17], it is the first protocol to achieve the lower bound on Fast Paxos [15] quorum sizes, and it corrects previously undiscovered errors in DPaxos [26].

- **Generality.** Reconfiguration protocols are hard to invent, so ideally we would re-use existing protocols whenever possible. MultiPaxos' horizontal reconfiguration [18] is arguably the most popular reconfiguration technique, but it makes fundamental assumptions that make it incompatible with many existing replication protocols. Matchmaker Paxos does not makes these restrictive assumptions, and can be more easily used by other replication protocols (Section 7).

- **Proven Safe.** We describe Matchmaker Paxos and Matchmaker MultiPaxos precisely and prove that both are safe (Sections 3, 4, 5, 6). Unfortunately, this is not often done for reconfiguration protocols [23, 30, 32, 31].

In a nutshell, our protocols work by leveraging two key design ideas. The first is to *decouple reconfiguration* from the standard processing path. Many replication protocols [18, 28] have nodes that are responsible for both processing commands and for orchestrating reconfigurations. By contrast, Matchmaker Paxos introduces a set of distinguished matchmaker nodes that are solely responsible for managing reconfigurations. This decoupling, along with a number of novel protocol optimizations, allow us to perform reconfiguration quickly in the background, without degrading performance.

The second design point is to *reconfigure across rounds, not commands*. Replication protocols based on classical MultiPaxos assume a totally ordered log of chosen commands, and reconfigure across log entries: each log entry is handled by a single configuration. Matchmaker Paxos instead works *across rounds of consensus*, as proposed by Vertical Paxos [17]. Different Paxos rounds—even for the same command—can use different configurations. Many replication protocols in fact have no sequential log, but almost every one has some form of rounds (a.k.a. terms, epochs, views). Matchmaker Paxos gets its generality from this design point, and goes beyond Vertical Paxos in terms of simplicity and efficiency (Section 7).

2. BACKGROUND

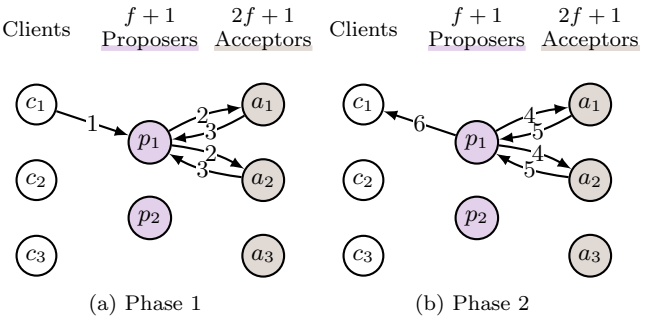
2.1 System Model

Throughout the paper, we assume an asynchronous network model in which messages can be arbitrarily dropped, delayed, and reordered. We assume machines can fail by crashing but do not act maliciously; i.e., we do not consider Byzantine failures. We assume that machines operate at arbitrary speeds, and we do not assume clock synchronization. We assume a discovery service that nodes can use to find each other, but do not require that this service be strongly consistent. A node can safely communicate with outdated nodes. A system like DNS would suffice. Every protocol discussed in this paper assumes that at most f machines will fail for some configurable f.

2.2 Paxos

A **consensus protocol** is a protocol that selects a single value from a set of proposed values. **Paxos** [12, 13] is one of the oldest and most popular consensus protocols. We assume that the reader is familiar with Paxos, but we review the parts of the protocol that are most important to understand for the rest of the paper.

A Paxos deployment that tolerates f faults consists of an arbitrary number of clients, $f+1$ nodes called **proposers**, and $2f+1$ nodes called **acceptors**, as illustrated in Figure 1. To reach consensus on a value, an execution of Paxos is divided into a number of rounds, each round having two phases: Phase 1 and Phase 2. Every round is orchestrated by a single pre-determined proposer. The set of rounds can be any unbounded totally ordered set with a least element. It is common to let the set of rounds be the set of lexicographically ordered integer pairs (r, id) where r is an integer and id is a proposer id, where a proposer is responsible for executing every round that contains its id.



(a) Phase 1 (b) Phase 2

Figure 1: Paxos communication diagram ($f = 1$).

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

More concretely, Paxos executes as follows. When a client wants to propose a value x, it sends x to a proposer p. Upon receiving x, p begins executing one round of Paxos, say round i. First, it executes Phase 1. It sends $\textsc{Phase1A}\langle i \rangle$ messages to at least a majority of the acceptors. An acceptor ignores a $\textsc{Phase1A}\langle i \rangle$ message if it has already received a message in a larger round. Otherwise, it replies with a $\textsc{Phase1B}\langle i, vr, vv \rangle$ message containing the largest round vr in which the acceptor voted and the value it voted for, vv. If the acceptor hasn't voted yet, then $vr = -1$ and $vv = \mathsf{null}$. When the proposer receives Phase1B messages from a majority of the acceptors, Phase 1 ends and Phase 2 begins.

At the beginning of Phase 2, the proposer uses the Phase1B messages that it received in Phase 1 to select a value x such that no value other than x has been or will be chosen in any

round less than i. Specifically x is the vote value associated with the largest received vote round, or any value if no acceptor had voted (see [13] for details). Then, the proposer sends $\textsc{Phase2A}\langle i, x\rangle$ messages to at least a majority of the acceptors. An acceptor ignores a $\textsc{Phase2A}\langle i, x\rangle$ message if it has already received a message in a larger round. Otherwise, it votes for x and sends back a $\textsc{Phase2B}\langle i\rangle$ message to the proposer. If a majority of acceptors vote for the value (i.e. if the proposer receives $\textsc{Phase2B}\langle i\rangle$ messages from at least a majority of the acceptors), then the value is chosen, and the proposer informs the client. This execution is illustrated in Figure 1.

2.3 Flexible Paxos

Paxos deploys a set of $2f + 1$ acceptors, and proposers communicate with at least a *majority* of the acceptors in Phase 1 and in Phase 2. **Flexible Paxos** [7] is a Paxos variant that generalizes the notion of a *majority* to that of a *quorum*. More specifically, Flexible Paxos introduces the notion of a **configuration** $C = (A; P1; P2)$. A is a set of acceptors. $P1$ and $P2$ are sets of **quorums**, where each quorum is a subset of A. A configuration satisfies the property that every quorum in $P1$ (known as a **Phase 1 quorum**) intersects every quorum in $P2$ (known as a **Phase 2 quorum**).

Flexible Paxos is identical to Paxos with the exception that proposers now communicate with an arbitrary Phase 1 quorum in Phase 1 and an arbitrary Phase 2 quorum in Phase 2. In the remainder of this paper, we assume that all protocols operate using quorums from an arbitrary configuration rather than majorities from a fixed set of $2f + 1$ acceptors. So when we say Paxos, we really mean Flexible Paxos, and when we say MultiPaxos, we really mean Flexible MultiPaxos. Note that configurations are also known as quorum systems [35].

3. MATCHMAKER PAXOS

We now present **Matchmaker Paxos**, a Paxos variant that allows us to reconfigure the set of acceptors. After we build some intuition about the protocol, we describe the protocol in detail and prove it is safe. In the next section, we'll extend Matchmaker Paxos to Matchmaker MultiPaxos.

3.1 Overview and Intuition

Matchmaker Paxos is largely identical to Paxos. Like Paxos, a Matchmaker Paxos deployment includes an arbitrary number of clients, a set of at least $f + 1$ proposers, and some set of acceptors, as illustrated in Figure 2. Paxos assumes that a *single, fixed* configuration of acceptors is used for every round. The big difference between Paxos and Matchmaker Paxos is that Matchmaker Paxos allows every round to have a *different* configuration of acceptors. Round 0 may use some configuration C_0, while round 1 may use some completely different configuration C_1. This idea was first introduced by Vertical Paxos [17].

Recall from Section 2, that a Paxos proposer in round i executes Phase 1 in order to (1) learn of any value that may have been chosen in a round less than i and (2) prevent any new values from being chosen in any round less than i. To do so, the proposer contacts the *fixed set* of acceptors. A Matchmaker Paxos proposer is no different. It must also execute Phase 1 and establish that these two properties hold.

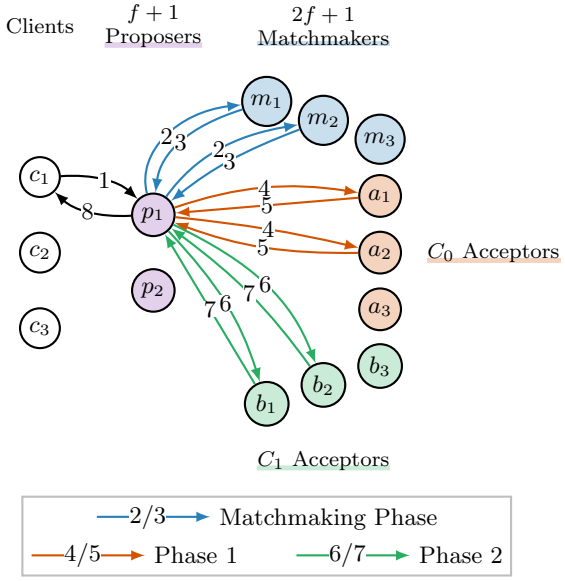


Figure 2: Matchmaker Paxos communication diagram.

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

However, every round uses a different configuration of acceptors, so how does the proposer of round i know which acceptors to contact in Phase 1? To resolve this question, a Matchmaker Paxos deployment also includes a set of $2f + 1$ **matchmakers**. When a proposer begins executing round i, it selects a configuration C_i. It sends the configuration C_i to the matchmakers, and the matchmakers reply with the configurations used in previous rounds. We call this the Matchmaking phase. The proposer then executes Phase 1 of Paxos with these prior configurations, and then executes Phase 2 with configuration C_i, as illustrated in Figure 2. Naively, the extra round trip of communication with the matchmakers and the large number of configurations in Phase 1 make Matchmaker Paxos slow. In Sections 3.4, 4.4, and 5, we'll introduce optimizations to eliminate these costs.

3.2 Details

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

When a matchmaker receives a $\textsc{MatchA}\langle i, C_i\rangle$ message, it checks to see if it had previously received a $\textsc{MatchA}\langle j, C_j\rangle$ message for some round $j \geq i$. If so, the matchmaker ignores the $\textsc{MatchA}\langle i, C_i\rangle$ message. Otherwise, it inserts C_i in log entry i and computes the set H_i of previous configurations in the log: $H_i = \{(j, C_j) \mid j < i, C_j \in L\}$. It then replies to the proposer with a $\textsc{MatchB}\langle i, H_i\rangle$ message. An example execution of a matchmaker is illustrated in Figure 3. Matchmaker pseudocode is given in Algorithm 1.

When the proposer in round i receives $\textsc{MatchB}\langle i, H_i^1\rangle$, ..., $\textsc{MatchB}\langle i, H_i^{f+1}\rangle$ from $f+1$ matchmakers, it computes

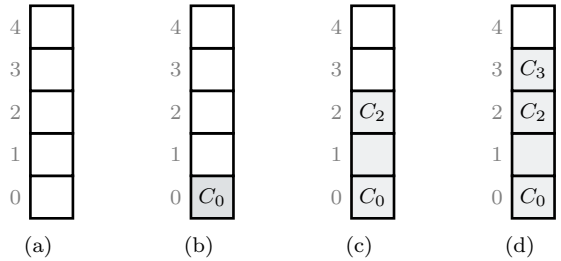


Figure 3: A matchmaker's log over time. (a) Initially, the matchmaker's log is empty. (b) Then, the matchmaker receives MatchA$\langle 0, C_0\rangle$. It inserts C_0 in log entry 0 and returns MatchB$\langle 0, \emptyset\rangle$ since the log does not contain any configuration in any round less than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

Algorithm 1 Matchmaker Pseudocode

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

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

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

Acceptor and proposer pseudocode are given in Algorithm 2 and Algorithm 3 respectively. To keep things simple, we assume that round numbers are integers, but generalizing is straightforward. A Matchmaker Paxos acceptor is identical to a Paxos acceptor. A Matchmaker Paxos proposer is nearly identical to a regular Paxos proposer with the exception of the Matchmaking phase and the configurations used in Phase 1 and Phase 2. For clarity of exposition, we omit straightforward details surrounding re-sending dropped messages and nacking ignored messages.

3.3 Proof of Safety

We now prove that Matchmaker Paxos is safe; i.e. every execution of Matchmaker Paxos chooses at most one value. We include the proof not only to convince you that the protocol is safe, but also because we believe that the proof makes it clear how and why Matchmaker Paxos works the way it does.

Algorithm 2 Acceptor Pseudocode

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

Algorithm 3 Proposer Pseudocode

State: a value x, initially null
State: a round i, initially -1
State: the configuration C_i for round i, initially null
State: the prior configurations H_i for round i, initially null
1: **upon** receiving value y from a client **do**
2: $i \leftarrow$ next largest round owned by this proposer
3: $x \leftarrow y$
4: $C_i \leftarrow$ an arbitrary configuration
5: send MatchA$\langle i, C_i\rangle$ to all of the matchmakers
6: **upon** receiving MatchB$\langle i, H_i^1\rangle, \ldots,$ MatchB$\langle i, H_i^{f+1}\rangle$ from $f + 1$ matchmakers **do**
7: $H_i \leftarrow \bigcup_{j=1}^{f+1} H_i^j$
8: send Phase1A$\langle i\rangle$ to every acceptor in H_i
9: **upon** receiving Phase1B$\langle i, -, -\rangle$ from a Phase 1 quorum from every configuration in H_i **do**
10: $k \leftarrow$ the largest vr in any Phase1B$\langle i, vr, vv\rangle$
11: $v \leftarrow$ the corresponding vv in round k
12: **if** $k \neq -1$ **then**
13: send Phase2A$\langle i, v\rangle$ to every acceptor in C_i
14: **else**
15: send Phase2A$\langle i, x\rangle$ to every acceptor in C_i
16: **upon** receiving Phase2B$\langle i\rangle$ from a Phase 2 quorum **do**
17: x is chosen, inform the client

Proof. Our proof is based off of the Paxos safety proof in [15]. We prove, for every round i, the statement $P(i)$ which states that if a proposer proposes a value v in round i (i.e. sends a Phase2A message for value v in round i), then no value other than v has been or will be chosen in any round less than i. $P(i)$ suffices to prove that Matchmaker Paxos is safe for the following reason. Assume for contradiction that Matchmaker Paxos chooses distinct values x and y in rounds i and j with $i < j$. Some proposer must have proposed y in round j, so $P(j)$ ensures us that no value other than y could have been chosen in round i. But, x was chosen, a contradiction.

We prove $P(i)$ by strong induction on i. $P(0)$ is vacuous because there are no rounds less than 0. For the general case $P(i)$, we assume $P(0), \ldots, P(i-1)$. We perform a case analysis on the proposer's pseudocode. Either k is -1 or it is not (line 10). First, assume it is not. In this case, the proposer proposes v, the value proposed in round k (line 13). We must show that no value other than v has been or will

be chosen in any round $j < i$. We perform a case analysis on j.

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

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

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

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

Case 3: $j < k$**.** This case follows by induction from $P(k)$.

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

3.4 Optimizations

Optimization 1: Proactive Matchmaking. Note that the Matchmaking phase is independent of any client value, so a proposer can proactively execute the Matchmaking phase in round i *before* it receives a value from a client. This is similar to proactively executing Phase 1, a standard optimization.

Optimization 2: Phase 1 Bypassing. Assume that the proposer in round i has proactively executed the Matchmaking phase and Phase 1. Through Phase 1, it finds that $k = -1$ and thus learns that no value has been chosen in any round less than i (see the safety proof above). Assume that before executing Phase 2, the proposer transitions from round i to round $i+1$, which it also owns. After executing the Matchmaking phase in round $i+1$, the proposer can skip Phase 1 and proceed directly to Phase 2. Why? The proposer established in round i that no value has been or will be chosen in any round less than i. Moreover, because it did not run Phase 2 in round i, it also knows that no value has been or will be chosen in round i. Together, these imply that no value has been or will be chosen in any round less than $i+1$. Normally, the proposer would run Phase 1 in round $i+1$ to establish this fact, but since it has already established it, it can instead proceed directly to Phase 2.

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

$$(0,a,0) < (0,a,1) < (0,a,2) < (0,a,3) < \cdots$$
$$(0,b,0) < (0,b,1) < (0,b,2) < (0,b,3) < \cdots$$
$$(1,a,0) < (1,a,1) < (1,a,2) < (1,a,3) < \cdots$$

With this set of rounds, the proposer p in round (r,p,s) always owns the next round $(r,p,s+1)$.

We acknowledge that, for now, this optimization sounds very esoteric. For Matchmaker Paxos, it is. In the next section, we'll see how to apply the optimization to Matchmaker MultiPaxos. There, this optimization is essential for good performance.

Optimization 3: Garbage Collection. A proposer performs Phase 1 with every configuration returned by the matchmakers. This can be prohibitively expensive if the matchmakers return a large number of configurations. Later in Section 5, we introduce a garbage collection protocol to delete old configurations. In Section 8, we see empirically that Matchmakers usually return just a single configuration.

4. MATCHMAKER MULTIPAXOS

In this section, we extend Matchmaker Paxos to Matchmaker MultiPaxos.

4.1 MultiPaxos

First, we summarize MultiPaxos. Whereas Paxos is a consensus protocol that agrees on a single value, **MultiPaxos** [12, 34] is a **state machine replication protocol** that agrees on a sequence, or "log" of values. MultiPaxos manages multiple **replicas** of a state machine. Clients send state machine commands to MultiPaxos, MultiPaxos places the commands in a totally ordered log, and state machine replicas execute the commands in log order. By beginning in the same initial state and executing the same commands in the same order, all state machine replicas are kept in sync.

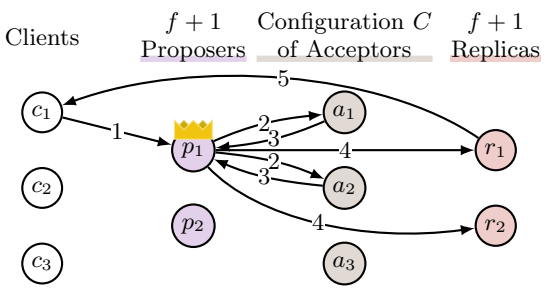


Figure 4: An example execution of MultiPaxos ($f = 1$). The leader is adorned with a crown.

To agree on a log of commands, MultiPaxos implements one instance of Paxos for every log entry. The ith instance of Paxos chooses the command in log entry i. More concretely, a MultiPaxos deployment that tolerates f faults consists of an arbitrary number of clients, at least $f+1$ proposers, a configuration C of acceptors, and at least $f+1$ replicas, as illustrated in Figure 4.

One of the proposers is elected leader in some round, say round i. We assume the leader knows that log entries up to and including log entry w have already been chosen (e.g., by communicating with previous leaders, or by communicating with the replicas). The leader then runs Phase 1 of Paxos in round i for *every* log entry larger than w. Note that even though there are an infinite number of log entries larger than w, the leader can execute Phase 1 using a finite amount of information. In particular, the leader sends a single Phase1A$\langle i \rangle$ message that acts as the Phase1A message for every log entry larger than w. Also, an acceptor replies with a Phase1B$\langle i, vr, vv \rangle$ message only for log entries in which the acceptor has voted. The infinitely many log entries in which the acceptor has not yet voted do not yield an explicit Phase1B message.

The leader's knowledge about the log after Phase 1 is shown in Figure 5. The leader knows that a prefix of the log (up to and including log entry w) has already been chosen. Then, there is a subsequence of the log that contains commands that may have already been chosen in a round less than i. This subsequence may contain unchosen entries, which we call "holes". Finally, the log has an infinite tail of empty entries in which the leader knows no command has previously been chosen.

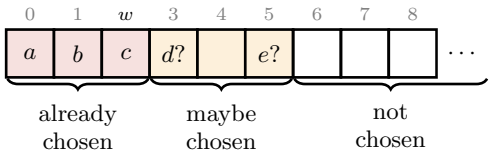


Figure 5: A leader's knowledge of the log after Phase 1.

After Phase 1, the leader sends a Phase2A message for every log entry in the middle subsequence of potentially chosen commands, proposing a "no-op" command for the holes. Simultaneously, the leader begins accepting client requests. When a client wants to propose a state machine command, it sends the command to the leader. The leader assigns log entries to commands in increasing order, beginning at the start of the infinite tail of unchosen entries. It then runs Phase 2 of Paxos to get the command chosen in that entry in round i. Once the leader learns that a command has been chosen in a given log entry, it informs the replicas. Replicas insert chosen commands into their logs and execute the logs in prefix order, sending the results of execution back to the clients. This execution is illustrated in Figure 4.

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

4.2 Matchmaker MultiPaxos

We first extend Matchmaker Paxos to Matchmaker MultiPaxos with Optimization 1 (Proactive Matchmaking) but without Optimization 2 (Phase 1 Bypassing) or 3 (Garbage Collection). We'll see how to incorporate Optimization 2 momentarily and Optimization 3 in the next section. The extension from Matchmaker Paxos to Matchmaker MultiPaxos is completely analogous to the extension of Paxos to MultiPaxos. Matchmaker MultiPaxos reaches consensus on a totally ordered log of state machine commands, one log entry at a time, using one instance of Matchmaker Paxos for every log entry.

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

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

The leader then enters Phase 2 and operates exactly as it would in MultiPaxos. It executes Phase 2 with C_i for any potentially unchosen values it learned about in Phase 1, and it fills in any holes in the log with no-ops. Moreover, when it receives a state machine command from a client, it assigns the command a log entry, runs Phase 2 with the acceptors in C_i, and informs the replicas when the command is chosen. Replicas execute commands in log order and send the results of executing commands back to the clients.

4.3 Discussion

Note that unlike regular MultiPaxos, Matchmaker MultiPaxos does not require a separate mechanism to perform a reconfiguration. Reconfiguration is baked into the protocol. To change from some configuration C_{old} in round i to some new configuration C_{new}, the leader of round i simply advances to round $i+1$ and selects the new configuration C_{new}. The new configuration is active immediately after the Matchmaking phase, a one round trip delay. Though the new configuration is active immediately, it is not safe to deactivate the acceptors in the old configuration immediately. They are still needed. In Section 5, we'll see why this is the case and when it is safe to retire old configurations.

Also note that Matchmaker MultiPaxos does *not* perform the Matchmaking Phase or Phase 1 on the critical path of normal execution. Similar to how MultiPaxos only executes Phase 1 once per leader change (and not once per command), Matchmaker MultiPaxos runs the Matchmaking phase and Phase 1 only during a leader change. In the normal case (i.e. during Phase 2), Matchmaker MultiPaxos and MultiPaxos are identical. Matchmaker MultiPaxos does not introduce

any overheads during normal case execution.

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

4.4 Optimization

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

This is illustrated in Figure 6. Figure 6 shows a leader p_1 reconfiguring from a configuration of acceptors C_{old} consisting of acceptors a_1, a_2, and a_3 in round i to a new configuration of acceptors C_{new} consisting of acceptors b_1, b_2, and b_3 in round $i+1$. While the leader performs the reconfiguration, clients continue to send state machine commands to the leader. We consider such a command and perform a case analysis on when the command arrives at the leader to see whether or not the command has to be stalled.

Case 1: Matchmaking (Figure 6a). If the leader receives a command during the Matchmaking phase, then the leader can process the command as normal in round i using the acceptors in C_{old}. Even though the leader is executing the Matchmaking phase in round $i+1$ and is communicating with the matchmakers, the acceptors in C_{old} are oblivious to this and can process commands in Phase 2 in round i as usual.

Case 2: Phase 1 (Figure 6b). If the leader receives a command during Phase 1, then the leader cannot process the command. It must delay the processing of the command until Phase 1 finishes. Here's why. Once an acceptor in C_{old} receives a Phase1A$\langle i+1\rangle$ message, it will reject any future commands in rounds less than $i+1$, so the leader is unable to send the command to C_{old}. The leader also cannot send the command to C_{new} in round $i+1$ because it has not yet finished executing Phase 1.

Case 3: Phase 2 (Figure 6c). If the leader receives a command during Phase 2, then the leader can send the command to the new acceptors in C_{new} in round $i+1$. This is the normal case of execution.

In summary, any commands received during Phase 1 of a reconfiguration are delayed. Fortunately, we can eliminate this problem by applying Optimization 2 (Phase 1 Bypassing) from Section 3 to Matchmaker MultiPaxos. Consider a leader performing a reconfiguration from C_i in round i to C_{i+1} in round $i+1$. At the end of the Matchmaking phase and at the beginning of Phase 1 (in round $i+1$), let k be the largest log entry that the leader has assigned to a command. That is, all log entries after entry k are empty. These log entries satisfy the preconditions of Optimization 2, so it is safe for the leader to bypass Phase 1 in round $i+1$ for these log entries in the following way. When a leader receives a command after the Matchmaking phase, it assigns the command a log entry larger than k, skips Phase 1, and executes Phase 2 in round $i+1$ with C_{new} immediately.

With this optimization and the round assignment described in Section 3.4, no state machine commands are delayed. Commands received during the Matchmaking phase or earlier are chosen in round i by C_{old} in log entries up to and including k. Commands received during Phase 1, Phase 2, or later are chosen in round $i+1$ by C_{new} in log entries $k+1$, $k+2$, $k+3$, and so on. With this optimization Matchmaker MultiPaxos can be reconfigured with minimal performance degradation. We confirm this empirically in Section 8.

5. RETIRING OLD CONFIGURATIONS

We've discussed how to introduce new configurations. Now, we explain how to shut down old configurations. We begin with Matchmaker Paxos and then generalize to Matchmaker MultiPaxos.

5.1 Matchmaker Paxos (How)

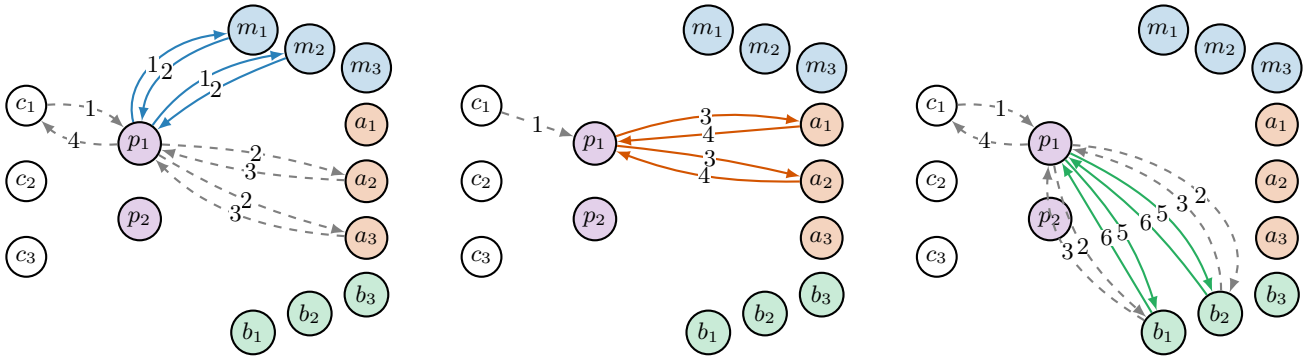
At the beginning of round i, a proposer p executes the Matchmaking phase and computes a set H_i of configurations in rounds less than i. The proposer then executes Phase 1 with the acceptors in these configurations. Assume H_i contains a configuration C_j for a round $j < i$. If we prematurely shut down the acceptors in C_j, then proposer p will get stuck in Phase 1, waiting for Phase1B messages from a quorum of nodes that have been shut down. Therefore, we cannot shut down the acceptors in a configuration C_j until we are sure that the matchmakers will never again return C_j during the Matchmaking phase.

Thus, we extend Matchmaker Paxos to allow matchmakers to garbage collect configurations from their logs, ensuring that the garbage collected configurations will not be returned during any future Matchmaking phase. More concretely, a proposer p can now send a GarbageA$\langle i\rangle$ command to the matchmakers informing them to garbage collect all configurations in rounds less than i. When a matchmaker receives a GarbageA$\langle i\rangle$ message, it deletes log entry $L[j]$ for every round $j < i$. It then updates a garbage collection watermark w to the maximum of w and i and sends back a GarbageB$\langle i\rangle$ message to the proposer. This is shown in Algorithm 4.

Algorithm 4 Matchmaker Pseudocode (with GC). Changes to Algorithm 1 are shown in red.

```
State: a log L indexed by round, initially empty
State: a garbage collection watermark w, initially 0
upon receiving GarbageA⟨i⟩ from proposer p do
    for all j < i do
        delete L[j]
    w ← max(w, i)
    send GarbageB⟨i⟩ to p
upon receiving MatchA⟨i, C_i⟩ from proposer p do
    if i < w or ∃ C_j in round j ≥ i in L then
        ignore the MatchA⟨i, C_i⟩ message
    else
        H_i ← {(j, C_j) | C_j ∈ L}
        L[i] ← C_i
        send MatchB⟨i, w, H_i⟩ to p
```

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



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

Figure 6: An example Matchmaker MultiPaxos reconfiguration without Optimization 2. The leader p_1 reconfigures from the acceptors a_1, a_2, a_3 to the acceptors b_1, b_2, b_3. Client commands are drawn as gray dashed lines. For simplicity, we assume that every proposer also serves as a replica.

computes $w = \max_{j=1}^{f+1} w_j$ and prunes every configuration in H_i in a round less than w. In other words, if any of the $f+1$ matchmakers have garbage collected round j, then the proposer also garbage collects round j.

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

Thus, once a configuration has been garbage collected by a majority of the matchmakers, we can shut down the acceptors in the configuration.

5.2 Matchmaker Paxos (When)

Once a configuration has been garbage collected, it is safe to shut it down, but when is it safe to garbage collect a configuration? It is certainly not always safe. For example, if we prematurely garbage collect configuration C_j in round j, a future proposer in round $i > j$ may not learn about a value v chosen in round j and then erroneously get a value other than v chosen in round i. There are three situations in which it is safe for a proposer p_i in round i to issue a GarbageA$\langle i \rangle$ command. We explain the three situations and provide intuition on why they are safe. We refer the reader to the technical report for a proof that formalizes the intuition [36].

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

Scenario 2. If the proposer p_i executes Phase 1 in round i and finds $k = -1$, then it can safely issue a GarbageA$\langle i \rangle$ command. Recall from Section 3 that if $k = -1$, then no value has been or will be chosen in any round less than i. This situation is similar to Scenario 1. Any future proposer p_j in round $j > i$ does not have to redundantly communicate with the configurations in rounds less than i since p_i already established that no value has been chosen in these rounds.

Scenario 3. If the proposer p_i learns that a value x has already been chosen and has been stored on $f+1$ other machines, then the proposer can safely issue a GarbageA$\langle i \rangle$ command after it informs a Phase 2 quorum of acceptors in C_i of this fact. Any future proposer p_j in round $j > i$ will contact a Phase 1 quorum of C_i and encounter at least one acceptor that knows the value x has already been chosen. When this acceptor informs p_j that a value x has already been chosen, p_j stops executing the protocol entirely and simply fetches the value x from one of the $f+1$ machines that store the value.

5.3 Matchmaker MultiPaxos

Recall that the Matchmaker MultiPaxos leader p_i in round i uses a single configuration C_i for *every* log entry. The leader p_i can safely issue a GarbageA$\langle i \rangle$ command to the matchmakers once it ensures that *every* log entry satisfies one of the three scenarios listed above. It does so as follows.

Recall from Figure 5 that at the end of Phase 1 and at the beginning of Phase 2, the log can be divided into three regions: a prefix of log entries that the leader knows have already been chosen, a subsequence of commands that the leader gets chosen in Phase 2, and an infinite tail of empty entries. Scenario 2 applies to the infinite tail of empty log entries. These are the log entries for which $k = -1$. Scenario 1 applies to the middle subsequence of commands once the leader gets them chosen. Scenario 3 applies to the prefix of chosen entries if we make the following adjustments. First, we deploy $2f+1$ replicas instead of $f+1$. Second, the leader ensures that the prefix of previously chosen log entries are stored on at least $f+1$ of the $2f+1$ replicas (this ensures that despite f replica failures, some replica will store the values). Third, the leader informs a Phase 2 quorum of C_i acceptors that these commands have been stored on the replicas.

In summary, the leader p_i of round i executes as follows. It executes the Matchmaking phase yielding prior configurations H_i. It then executes Phase 1 with the configurations in H_i. It enters Phase 2 and gets the middle subsequence of commands chosen. It also informs a Phase 2 quorum of C_i acceptors once the prefix of previously chosen com-

mands have been stored on $f+1$ replicas. It then issues a GarbageA$\langle i \rangle$ command to the matchmakers and awaits $f+1$ GarbageB$\langle i \rangle$ responses. At this point, all previous configurations can be shut down.

Note that the leader can begin processing state machine commands from clients as soon as it enters Phase 2. It does not have to stall client commands during garbage collection. Note also that during normal operation (i.e. operation with a single stable leader), old configurations are garbage collected after a few round trips of communication. In Section 8, we find that H_i almost always contains only a single configuration (i.e. C_{i-1}).

6. RECONFIGURING MATCHMAKERS

Thus far, we have discussed how Matchmaker MultiPaxos allows us to reconfigure the set of acceptors. In this section, we discuss how to reconfigure proposers, replicas, and matchmakers.

Reconfiguring proposers and replicas is straightforward. In fact, Matchmaker MultiPaxos reconfigures proposers and replicas in exactly the same way as MultiPaxos [34], so we do not discuss it at length. In short, a proposer can be safely added or removed at any time. Replicas can also be safely added or removed at any time so long as we ensure that commands replicated on $f+1$ replicas remain replicated on $f+1$ replicas. For performance, a newly introduced proposer should contact an existing proposer or replica to learn about the prefix of already chosen commands, and a newly introduced replica should copy the log from an existing replica.

Reconfiguring matchmakers is a bit more involved, but still relatively straightforward. Recall that Matchmaker MultiPaxos performs the Matchmaking phase only during a leader change. Thus, for the vast majority of the time—specifically, when there is a single, stable leader—the matchmakers are idle. When idle, they don't send or receive any messages at all and are completely off the critical path of command processing. This means that the way we reconfigure the matchmakers has to be safe, but it doesn't have to be efficient. The matchmakers can be reconfigured at any time between leader changes without any impact on the performance of the protocol.

Thus, we use the simplest approach to reconfiguration: we shut down the old matchmakers and replace them with new ones, making sure that the new matchmakers' initial state is the same as the old matchmakers' final state. More concretely, we reconfigure from a set M_{old} of matchmakers to a new set M_{new} as follows. First, a proposer (or any other node, it doesn't really matter) sends a StopA$\langle\rangle$ message to the matchmakers in M_{old}. When a matchmaker m_i receives a StopA$\langle\rangle$ message, it stops processing messages (except for other StopA$\langle\rangle$ messages) and replies with StopB$\langle L_i, w_i \rangle$ where L_i is m_i's log and w_i is its garbage collection watermark. When the proposer receives StopB messages from $f+1$ matchmakers, it knows that the matchmakers have effectively been shut down. It computes w as the maximum of every returned w_i. It computes L as the union of the returned logs, and removes all entries of L that appear in a round less than w. An example of this log merging is illustrated in Figure 7.

The proposer then sends L and w to all of the matchmakers in M_{new}. Each matchmaker adopts these values as its initial state. At this point, the matchmakers in M_{new} *cannot* begin processing commands yet. Naively, it is possible that two different nodes could simultaneously attempt to reconfigure to two disjoint sets of matchmakers, say M_{new} and M'_{new}. To avoid this, every matchmaker in M_{old} doubles as a Paxos acceptor. A proposer attempting to reconfigure to M_{new} acts as a Paxos proposer and gets the value M_{new} chosen by the matchmakers (which are acting as Paxos acceptors). Once M_{new} is chosen, the proposer notifies the matchmakers in M_{new} that the reconfiguration is complete and that they are free to start processing commands. A safety proof for Matchmaker MultiPaxos with matchmaker reconfiguration is given in the technical report [36].

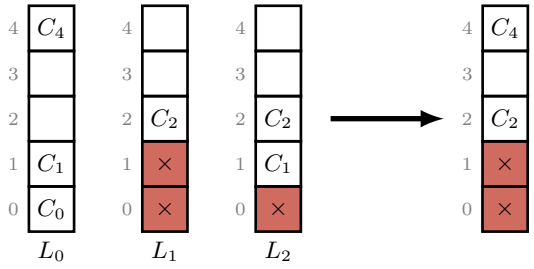


Figure 7: An example of merging three matchmaker logs (L_0, L_1, and L_2) during a matchmaker reconfiguration. Garbage collected log entries are shown in red.

7. INSIGHTS AND GENERALITY

In this section, we discuss some theoretical insights that Matchmaker Paxos provides, and we describe how to extend the ideas from Matchmaker Paxos to other replication protocols.

7.1 Insights

Vertical Paxos Matchmaker Paxos was inspired by Vertical Paxos [17], but differs in ways that improve simplicity, efficiency, garbage collection, and practicality.

In terms of simplicity, Vertical Paxos requires an external master, which is itself implemented using state machine replication. Our matchmakers are analogous to the external master but show that such a master does not require a nested invocation of state machine replication.

From an efficiency perspective, Vertical Paxos requires that a proposer execute Phase 1 in order to perform a reconfiguration. Thus, Vertical Paxos can not be extended to MultiPaxos without causing performance degradation during reconfiguration. This is not the case for matchmakers thanks to Phase 1 Bypassing.

Vertical Paxos garbage collects old configurations in situations similar to Scenario 1 and Scenario 2 from Section 5. It does not include Scenario 3, and as a result, it is unclear how to extend Vertical Paxos' garbage collection to MultiPaxos.

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

Fast Paxos. Fast Paxos [15] is a Paxos variant that shaves off one network delay from Paxos in the best case, but can have higher delays if concurrently proposed commands conflict. While Paxos quorums consist of $f+1$ out of $2f+1$ acceptors, Fast Paxos requires larger quorums, typically

$f + \lceil f/2 \rceil + 1$ nodes out of $2f + 1$ acceptors. Many protocols have reduced Fast Paxos quorum sizes a bit, but to date, Fast Paxos quorum sizes have remained larger than classic Paxos quorum sizes.

Using matchmakers, we can implement Fast Paxos with only $f + 1$ acceptors (and hence with $f + 1$-sized quorums). These small quorum sizes are covered by Fast Paxos' quorum intersection requirements [15, 8], but we are the first to show how to achieve them. Specifically, we deploy Fast Paxos with $f + 1$ acceptors, with a single unanimous Phase 2 quorum, and with singleton Phase 1 quorums. Fast Paxos leaders perform the Matchmaking phase before Phase 1, as with Matchmaker Paxos, but the protocol remains otherwise unchanged. A full description of the protocol and a proof of correctness is given in our technical report [36]. Note that Fast Paxos cannot leverage Phase 1 Bypassing.

DPaxos. Matchmaker Paxos subsumes DPaxos and corrects some previously undiscovered errors in the protocol. DPaxos is a Paxos variant that allows every round to use a different subset of acceptors from some fixed set of acceptors. Matchmaker Paxos obviates the need for a fixed set of nodes. DPaxos' scope is limited to a single instance of consensus, whereas Matchmaker MultiPaxos shows how to efficiently reconfigure across multiple instances of consensus simultaneously. In the process of developing Matchmaker MultiPaxos, we discovered DPaxos' garbage collection algorithm is incorrect, and can lead to multiple values being chosen within a single instance of consensus. Matchmaker MultiPaxos' garbage collection algorithm fixes these issues. A full description of the bug is given in our technical report [36].

7.2 Generality

First, we elaborate on MultiPaxos' horizontal reconfiguration. To reconfigure from a set of nodes N to a new set of nodes N', the MultiPaxos leader gets the value N' chosen in the log at some index i. All commands in the log starting at position $i + \alpha$ are chosen using the nodes in N' instead of the nodes in N, where α is some configurable parameter. The MultiPaxos leader can process at most α unchosen commands at a time, so typically, α is set very large. To avoid having to wait for α commands to be chosen before a reconfiguration takes effect, the MultiPaxos leader orchestrating the reconfiguration will often get a sequence of no-ops chosen in the log to artificially advance the log to index $i + \alpha$.

MultiPaxos' horizontal reconfiguration protocol is simple and has been proven correct. Unfortunately, it is fundamentally incompatible with replication protocols that do not have a log. Moreover, researchers are finding that avoiding a log can often be advantageous. For example, protocols like Generalized Paxos [14], EPaxos [24], Janus [25], BPaxos [37], and Caesar [1] arrange commands in a partially ordered graph instead of a totally ordered log to exploit commutativity between commands. CASPaxos [32] maintains a single value, instead of a log or graph, for simplicity. Databases like TAPIR [38] avoid ordering transactions in a log for improved performance, and databases like Meerkat [33] do the same to improve scalability.

Because these protocols do not have logs, they cannot use MultiPaxos' horizontal reconfiguration protocol. However, while none of the protocols have logs, *all* of them have rounds. This means that the protocols can either use Matchmaker Paxos directly, or at least borrow ideas from Matchmaker Paxos for reconfiguration. BPaxos, for example, uses Paxos as a subroutine. We can directly replace Paxos with Matchmaker Paxos. CASPaxos is almost identical to Paxos and can be extended to Matchmaker CASPaxos in the same way we extended Paxos to Matchmaker Paxos. These are two simple examples, and we don't claim that extending Matchmaker Paxos to some of the other more complicated protocols is always easy. But, the universality of rounds and the increasing rarity of logs makes Matchmaker Paxos an attractive foundation on top of which other protocols can build their own reconfiguration protocols.

8. EVALUATION

8.1 Reconfiguration

Experiment Description. Our implementation of Matchmaker MultiPaxos is available at `github.com/mwhittaker/matchmakers`. We implemented Matchmaker MultiPaxos in Scala using the Netty networking library. All communication is done using TCP. We deployed Matchmaker MultiPaxos on a number of m5.xlarge AWS EC2 instances within a single availability zone in the Northern California region. For a given f, we deployed $f + 1$ proposers, $2f + 1$ acceptors, $2f + 1$ matchmakers, and $2f + 1$ replicas. For simplicity, every node is deployed on its own machine, but in practice, nodes can be physically co-located. All of our results hold in a co-located deployment as well. For simplicity, we deploy Matchmaker MultiPaxos with a trivial no-op state machine in which every state machine command is a one byte no-op. All of our results generalize to more complex state machines as well (the choice of state machine is orthogonal to reconfiguration).

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

Results. The latency and throughput of Matchmaker MultiPaxos for $f = 1$ is shown in Figure 8. For space reasons, we report all of our results only for $f = 1$. The results for larger values of f are identical and shown in our technical report [36]. Throughput and latency are both computed using sliding one second windows. Median latency is shown using solid lines, while the 95% latency is shown as a shaded region above the median latency. The black vertical lines denote reconfigurations, and the red dashed vertical line denotes the acceptor failure.

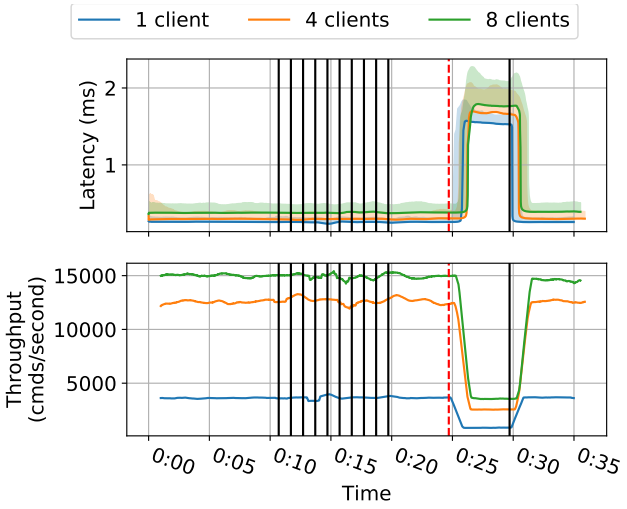


Figure 8: The latency and throughput of Matchmaker MultiPaxos with $f = 1$. The vertical black lines show reconfigurations. The vertical dashed red line shows an acceptor failure.

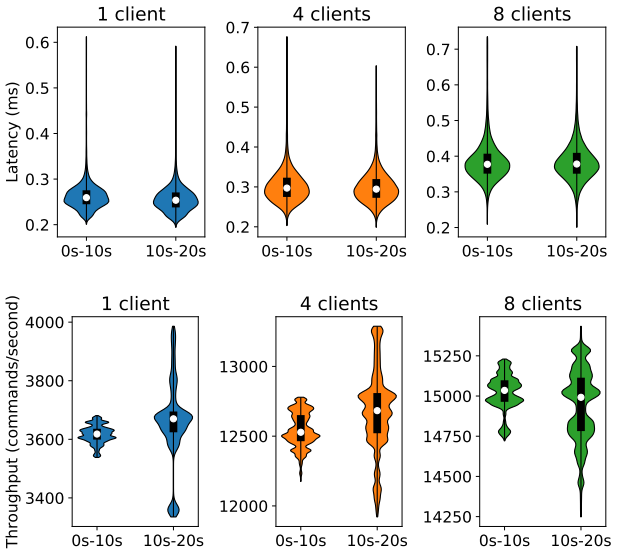


Figure 9: Violin plots of Figure 8 latency and throughput during the first 10 seconds and between 10 and 20 seconds.

Figure 9 includes violin plots of the protocol's latency and throughput (a) during the first 10 seconds and (b) between 10 and 20 seconds. The white circles show the median values, while the thick black rectangles show the 25th and 75th percentiles. The medians, interquartile ranges (IQR), and standard deviations of the latency and throughput are shown in Table 1.

For latency, reconfiguration has little to no impact (roughly 2% changes) on the medians, IQRs, or standard deviations. The violin plots confirm that the distribution of latencies are almost identical during the first 10 seconds and between 10 and 20 seconds. The one exception is that the 8 client standard deviation is significantly larger. This is due to a small number of outliers. Reconfiguration has little impact on median throughput, with all differences being statistically insignificant. The IQRs and standard deviations, however, do increase. The violin plots confirm that during a reconfiguration, there is more variation in throughput. Still, during a reconfiguration, the IQR is always less than 1% of the median throughput, and the standard deviation is always less than 4%.

For every reconfiguration, the new acceptors become active within a millisecond. The old acceptors are garbage collected within five milliseconds. We implement Matchmaker MultiPaxos with an optimization called thriftiness [24]—where Phase2A messages are initially sent to a randomly selected Phase 2 quorum—so the throughput and latency expectedly degrade after we fail an acceptor. After we replace the failed acceptor, throughput and latency return to normal within two seconds.

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

8.2 Leader Failure

Table 1: Figure 9 median, interquartile range, and standard deviation of latency and throughput.

Latency (ms)

	1 Client		4 Clients		8 Clients	
	0s-10s	10s-20s	0s-10s	10s-20s	0s-10s	10s-20s
median	0.260	0.254	0.298	0.295	0.378	0.379
IQR	0.016	0.018	0.027	0.026	0.030	0.032
stdev	0.070	0.060	0.091	0.085	0.100	0.334

Throughput (commands/second)

	1 Client		4 Clients		8 Clients	
	0s-10s	10s-20s	0s-10s	10s-20s	0s-10s	10s-20s
median	3,618	3,670	12,528	12,683	15,035	14,992
IQR	18	25	123	126	63	122
stdev	31	138	123	291	112	223

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

Results. The latency and throughput of the benchmarks are shown in Figure 10. During the first 7 seconds, throughput and latency are both stable. When the leader fails, the throughput expectedly drops to zero since all messages are processed by the leader (which has failed). We elect a new leader 5 seconds later. The leader does not reconfigure the acceptors—there is no need to since none of them have failed—but the leader still performs the Matchmaking phase. The throughput and latency return to normal within two seconds.

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

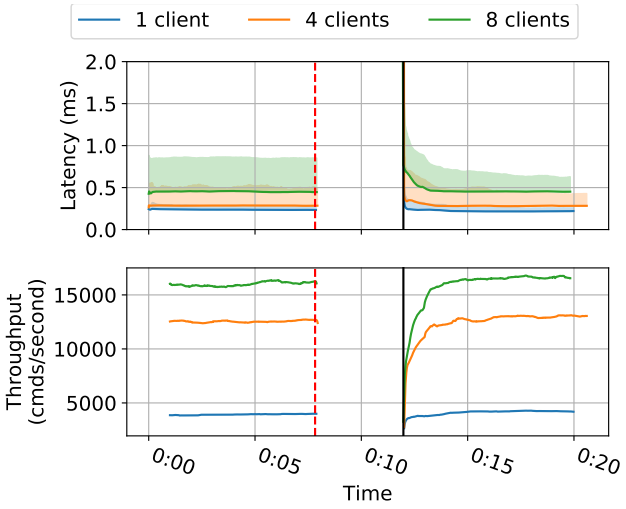


Figure 10: The latency and throughput of Matchmaker MultiPaxos with $f = 1$. The vertical dashed red line shows a leader failure.

Figure 11: The latency and throughput of Matchmaker MultiPaxos ($f = 1$). The dotted blue, dashed red, and vertical black lines show matchmaker reconfigurations, a matchmaker failure, and an acceptor reconfiguration respectively.

negligible.

8.3 Matchmaker Reconfiguration

Experiment Description. We deploy Matchmaker MultiPaxos as above. We again run three benchmarks with 1, 4, and 8 clients. Each benchmark runs for 40 seconds. During the first 10 seconds, there are no reconfigurations and no failures. Between 10 and 20 seconds, the leader reconfigures the set of matchmakers once every second. Every reconfiguration randomly selects $2f + 1$ matchmakers from a set of $2 \times (2f + 1)$ matchmakers. At 25 seconds, we fail a matchmaker. At 30 we perform a matchmaker reconfiguration to replace the failed matchmaker. At 35 seconds, we reconfigure the acceptors.

Results. The latency and throughput of Matchmaker MultiPaxos are shown in Figure 11. The latency and throughput of the protocol remains steady through the first ten matchmaker reconfigurations, through the matchmaker failure and recovery, and through the acceptor reconfiguration. The medians, IQRs, and standard deviations of the latency and throughput are very similar to the ones in Table 1. The full data can be found in our technical report [36].

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

9. RELATED WORK

In this section, we survey related work. See our technical report for a more thorough review [36].

MultiPaxos' Horizontal Reconfiguration. MultiPaxos' horizontal reconfiguration protocol [12, 18] uses the consensus it implements in order to reach consensus on a given configuration. This approach, also taken by systems such as Chubby [3, 4], does not require an extra reconfiguration algorithm or any additional nodes. Horizontal reconfiguration limits concurrency, as the proposer cannot have more than α outstanding operations. It can also be slow to reconfigure, as it needs to wait for α operations to be decided (though no-ops can help mitigate this). Unlike with Matchmaker Paxos, MultiPaxos cannot perform a reconfiguration if a proposer cannot contact a Phase 2 quorum of acceptors. Both MultiPaxos and Matchmaker MultiPaxos can perform a reconfiguration without disrupting the throughput or latency of the state machine. SMART [20] is a reconfiguration protocol that resolves many ambiguities in MultiPaxos' horizontal approach.

Raft. Raft [29] uses a reconfiguration protocol called joint consensus. Like MultiPaxos' horizontal reconfiguration, joint consensus is log-based and therefore incompatible with many existing replication protocols.

Viewstamped Replication (VR). VR [19] uses a stop-the-world approach to reconfiguration. During a reconfiguration, the entire protocol stops processing commands. Thus, while the reconfiguration is quite simple, it is inefficient. Stoppable Paxos [16] is similar to MultiPaxos' horizontal reconfiguration, but also uses a stop-the-world approach.

Fast Paxos Coordinated Recovery. Fast Paxos has an optimization called coordinated recovery that is similar to Phase 1 Bypassing. The main difference is that in coordinated recovery, a leader uses Phase 2 information in round i to skip Phase 1 in round $i + 1$, whereas with Phase 1 Bypassing, the leader instead uses Phase 1 information.

10. CONCLUSION

We presented Matchmaker Paxos and Matchmaker MultiPaxos to address the lack of research on the increasingly important topic of reconfiguration. Our protocols decouple reconfiguration from command processing and reconfigure over rounds instead of commands in order to achieve a number of desirable properties, both theoretical and practical. They can reconfigure without performance degradation, they provide insights into existing protocols, and they generalize better than existing techniques.

11. REFERENCES

[1] B. Arun, S. Peluso, R. Palmieri, G. Losa, and B. Ravindran. Speeding up consensus by chasing fast decisions. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 49–60. IEEE, 2017.

[2] M. Biely, Z. Milosevic, N. Santos, and A. Schiper. S-paxos: Offloading the leader for high throughput state machine replication. In *2012 IEEE 31st Symposium on Reliable Distributed Systems*, pages 111–120. IEEE, 2012.

[3] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI 06, page 335350, USA, 2006. USENIX Association.

[4] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC 07, page 398407, New York, NY, USA, 2007. Association for Computing Machinery.

[5] C. Chrysafis, B. Collins, S. Dugas, J. Dunkelberger, M. Ehsan, S. Gray, A. Grieser, O. Herrnstadt, K. Lev-Ari, T. Lin, et al. Foundationdb record layer: A multi-tenant structured datastore. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1787–1802, 2019.

[6] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Googles globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[7] H. Howard, D. Malkhi, and A. Spiegelman. Flexible paxos: Quorum intersection revisited. In *20th International Conference on Principles of Distributed Systems (OPODIS 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[8] H. Howard and R. Mortier. A generalised solution to distributed consensus. *arXiv preprint arXiv:1902.06776*, 2019.

[9] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. Mdcc: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 113–126, 2013.

[10] Kubernetes. Kubernetes. `https://kubernetes.io`. accessed: 2020-03-01.

[11] C. Labs. CockroachDB. `https://www.cockroachlabs.com/`. accessed: 2020-03-01.

[12] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[13] L. Lamport. Paxos made simple. 2001.

[14] L. Lamport. Generalized consensus and paxos. 2005.

[15] L. Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[16] L. Lamport, D. Malkhi, and L. Zhou. Stoppable paxos. *TechReport, Microsoft Research*, 2008.

[17] L. Lamport, D. Malkhi, and L. Zhou. Vertical paxos and primary-backup replication. 2009.

[18] L. Lamport, D. Malkhi, and L. Zhou. Reconfiguring a state machine. *SIGACT News*, 41(1):63–73, 2010.

[19] B. Liskov and J. Cowling. Viewstamped replication revisited. 2012.

[20] J. R. Lorch, A. Adya, W. J. Bolosky, R. Chaiken, J. R. Douceur, and J. Howell. The smart way to migrate replicated stateful services. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 103–115, 2006.

[21] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI08, page 369384, USA, 2008. USENIX Association.

[22] P. J. Marandi, M. Primi, N. Schiper, and F. 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.

[23] I. Moraru, D. G. Andersen, and M. Kaminsky. A proof of correctness for egalitarian paxos. Technical report, Technical report, Parallel Data Laboratory, Carnegie Mellon University, 2013.

[24] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 358–372, 2013.

[25] S. Mu, L. Nelson, W. Lloyd, and J. Li. Consolidating concurrency control and consensus for commits under conflicts. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 517–532, 2016.

[26] F. Nawab, D. Agrawal, and A. 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*, pages 1221–1236. ACM, 2018.

[27] D. Ongaro. Bug in single-server membership changes. `https://groups.google.com/forum/#!msg/raft-dev/t4xj6dJTP6E/d2D9LrWRza8J`. accessed: 2020-03-01.

[28] D. Ongaro. *Consensus: Bridging theory and practice*. PhD thesis, Stanford University, 2014.

[29] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 305–319, 2014.

[30] D. R. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 43–57, 2015.

[31] S. Rizvi, B. Wong, and S. Keshav. Canopus: A scalable and massively parallel consensus protocol. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, pages 426–438, 2017.

[32] D. Rystsov. Caspaxos: Replicated state machines without logs. *arXiv preprint arXiv:1802.07000*, 2018.

[33] A. Szekeres, M. Whittaker, J. Li, S. Naveen, A. Krishnamurthy, D. Ports, and I. Zhang. Meerkat:

Multicore-scalable replicated transactions following the zero-coordination principle. In *Proceedings of the Fourteenth EuroSys Conference 2020*, 2020.

[34] R. Van Renesse and D. Altinbuken. Paxos made moderately complex. *ACM Computing Surveys (CSUR)*, 47(3):1–36, 2015.

[35] M. Vukolić et al. The origin of quorum systems. *Bulletin of EATCS*, 2(101), 2013.

[36] M. Whittaker, N. Giridharan, A. Szekeres, J. M. Hellerstein, H. Howard, F. Nawab, and I. Stoica. Matchmaker paxos: A reconfigurable consensus protocol [technical report]. `https://mwhittaker.github.io/publications/matchmaker_paxos_tr.pdf`.

[37] M. Whittaker, N. Giridharan, A. Szekeres, J. M. Hellerstein, and I. Stoica. Bipartisan paxos: A modular state machine replication protocol. `https://mwhittaker.github.io/publications/compartmentalized_bipartisan_paxos.pdf`.

[38] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. Ports. Building consistent transactions with inconsistent replication. *ACM Transactions on Computer Systems (TOCS)*, 35(4):1–37, 2018.