

# Matchmaker Paxos: A Reconfigurable Consensus Protocol

Michael Whittaker<sup>1</sup>, Neil Giridharan<sup>1</sup>, Adriana Szekeres<sup>2</sup>, Joseph M. Hellerstein<sup>1</sup>,  
Heidi Howard<sup>3</sup>, Faisal Nawab<sup>4</sup>, Ion Stoica<sup>1</sup>

<sup>1</sup> University of California, Berkeley, <sup>2</sup> University of Washington, <sup>3</sup> University of  
Cambridge, <sup>4</sup> 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 [5], CockroachDB [11], and FoundationDB [4]—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 [27], 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 [23, 29, 31, 30], propose theoretically safe but inefficient reconfiguration protocols [13, 19], or do not discuss reconfiguration at all [14, 20, 21, 3, 2]. In recent years, state-of-the-art databases—like MDCC [9], Janus [24], and TAPIR [37]—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 [26].

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).

**Theoretical Insights.** Matchmaker Paxos offers theoretical insights into many existing protocols (Section 7). It extends Vertical Paxos [17], it is the first protocol to achieve the theoretical lower bound on Fast Paxos [14] quorum sizes,

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/4.0/>. For any use beyond those covered by this license, obtain permission by emailing [info@vldb.org](mailto:info@vldb.org). Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

*Proceedings of the VLDB Endowment*, Vol. 13, No. xxx  
ISSN 2150-8097.

DOI: <https://doi.org/10.14778/xxxxxxx.xxxxxxx>

and it corrects errors in DPaxos [25].

**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 assumptions that make it incompatible with many existing replication protocols. Matchmaker Paxos does not make 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 [22, 29, 31, 30].

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, 27] 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, 15] 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.

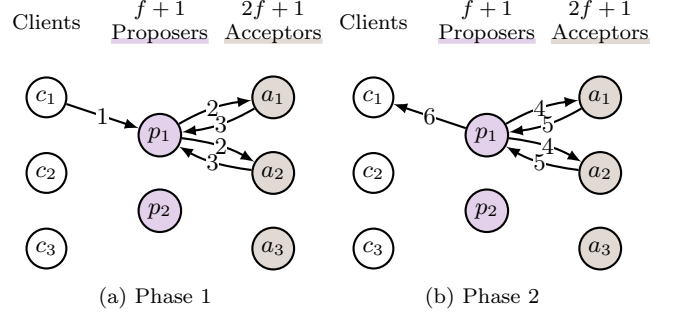


Figure 1: Paxos communication diagram ( $f = 1$ ).

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  $\text{PHASE1A}(i)$  messages to at least a majority of the acceptors. An acceptor ignores a  $\text{PHASE1A}(i)$  message if it has already received a message in a larger round. Otherwise, it replies with a  $\text{PHASE1B}(i, vr, vv)$  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  $vv = \text{null}$ . When the proposer receives  $\text{PHASE1B}$  messages from a majority of the acceptors, Phase 1 ends and Phase 2 begins.

At the start of Phase 2, the proposer uses the  $\text{PHASE1B}$  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 [15] for details). Then, the proposer sends  $\text{PHASE2A}(i, x)$  messages to at least a majority of the acceptors. An acceptor ignores a  $\text{PHASE2A}(i, x)$  message if it has already received a message in a larger round. Otherwise, it votes for  $x$  and sends back a  $\text{PHASE2B}(i)$  message to the proposer. If a majority of acceptors vote for the value

(i.e. if the proposer receives  $\text{PHASE2B}\langle i \rangle$  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*. 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, for example, we really mean Flexible Paxos. Note that configurations are also known as quorum systems [34].

## 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. 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

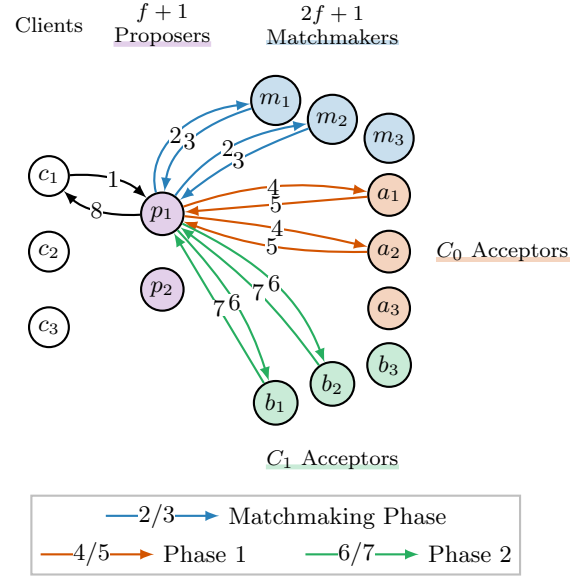


Figure 2: Matchmaker Paxos communication diagram.

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. It may also seem like we've solved the problem of reconfiguring acceptors by introducing a new problem of reconfiguring matchmakers, but this is not the case. In Section 6, we explain how we can reconfigure the matchmakers in a straightforward way.

### 3.2 Details

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

When a matchmaker receives a  $\text{MATCHA}\langle i, C_i \rangle$  message, it checks to see if it had previously received a  $\text{MATCHA}\langle j, C_j \rangle$  message for some round  $j \geq i$ . If so, the matchmaker ignores the  $\text{MATCHA}\langle i, C_i \rangle$  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  $\text{MATCHB}\langle i, H_i \rangle$  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  $\text{MATCHB}\langle i, H_i^1 \rangle, \dots, \text{MATCHB}\langle i, H_i^{f+1} \rangle$  from  $f + 1$  matchmakers, it computes  $H_i = \cup_{j=1}^{f+1} H_i^j$ . For example, with  $f = 1$  and  $i = 2$ , if the proposer in round 2 receives  $\text{MATCHB}\langle 2, \{(0, C_0)\} \rangle$  and  $\text{MATCHB}\langle 2, \{(1, C_1)\} \rangle$ , 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.

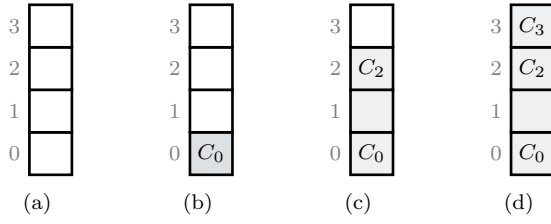


Figure 3: A matchmaker's log over time. (a) Initially, the matchmaker's log is empty. (b) Then, the matchmaker receives  $\text{MATCHA}\langle 0, C_0 \rangle$ . It inserts  $C_0$  in log entry 0 and returns  $\text{MATCHB}\langle 0, \emptyset \rangle$  since the log does not contain any configuration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

---

#### Algorithm 1 Matchmaker Pseudocode

---

**State:** a log  $L$  indexed by round, initially empty

```

1: upon receiving  $\text{MATCHA}\langle i, C_i \rangle$  from proposer  $p$  do
2:   if  $\exists$  a configuration  $C_j$  in round  $j \geq i$  in  $L$  then
3:     ignore the  $\text{MATCHA}\langle i, C_i \rangle$  message
4:   else
5:      $H_i \leftarrow \{(j, C_j) \mid C_j \in L\}$ 
6:      $L[i] \leftarrow C_i$ 
7:     send  $\text{MATCHB}\langle i, H_i \rangle$  to  $p$ 

```

---

The proposer then ends the Matchmaking phase and begins Phase 1. It sends  $\text{PHASE1A}$  messages to every acceptor in every configuration in  $H_i$  and waits to receive  $\text{PHASE1B}$  messages from at least a Phase 1 quorum from every configuration in  $H_i$ . Using the previous example, the proposer would send  $\text{PHASE1A}$  messages to every acceptor in  $C_0$  and  $C_1$  and would wait for  $\text{PHASE1B}$  messages from a Phase 1 quorum of  $C_0$  and a Phase 1 quorum of  $C_1$ . The proposer then runs Phase 2 with  $C_i$ .

Acceptor and proposer pseudocode are shown in Algorithm 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.

*Proof.* Our proof is based off of the Paxos safety proof in [14]. We prove, for every round  $i$ , the statement  $P(i)$ : “if a proposer proposes a value  $v$  in round  $i$  (i.e. sends a  $\text{PHASE2A}$  message for value  $v$  in round  $i$ ), then no value other than  $v$

---

#### 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  $\text{PHASE1A}\langle i \rangle$  from  $p$  with  $i > r$  do
2:    $r \leftarrow i$ 
3:   send  $\text{PHASE1B}\langle i, vr, vv \rangle$  to  $p$ 
4: upon receiving  $\text{PHASE2A}\langle i, x \rangle$  from  $p$  with  $i \geq r$  do
5:    $r, vr, vv \leftarrow i, i, x$ 
6:   send  $\text{PHASE2B}\langle i \rangle$  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  $\text{MATCHA}\langle i, C_i \rangle$  to all of the matchmakers
6: upon receiving  $\text{MATCHB}\langle i, H_i^1 \rangle, \dots, \text{MATCHB}\langle i, H_i^{f+1} \rangle$  from  $f+1$  matchmakers do
7:    $H_i \leftarrow \bigcup_{j=1}^{f+1} H_i^j$ 
8:   send  $\text{PHASE1A}\langle i \rangle$  to every acceptor in  $H_i$ 
9: upon receiving  $\text{PHASE1B}\langle i, -, - \rangle$  from a Phase 1 quorum from every configuration in  $H_i$  do
10:   $k \leftarrow$  the largest  $vr$  in any  $\text{PHASE1B}\langle i, vr, vv \rangle$ 
11:  if  $k \neq -1$  then
12:     $x \leftarrow$  the corresponding  $vv$  in round  $k$ 
13:    send  $\text{PHASE2A}\langle i, x \rangle$  to every acceptor in  $C_i$ 
14: upon receiving  $\text{PHASE2B}\langle i \rangle$  from a Phase 2 quorum do
15:    $x$  is chosen, inform the client

```

---

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  $P(0), \dots, P(i-1)$ . We perform a case analysis on the proposer's pseudocode. Either  $k$  is  $-1$  or it is not (line 11). First, assume it is not. In this case, the proposer proposes  $x$ , the value proposed in round  $k$  (line 12). We perform a case analysis on  $j$  to show that no value other than  $x$  has been or will be chosen in any round  $j < i$ .

**Case 1:**  $j > k$ . We show that no value has been or will be chosen in round  $j$ . Recall that at the end of the Matchmaking phase, the proposer computed the set  $H_i$  of prior configurations using responses from a set  $M$  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  $\text{PHASE1A}\langle i \rangle$  messages to all of the acceptors in  $C_j$ . A Phase 1 quorum of these acceptors, say  $Q$ , all received  $\text{PHASE1A}\langle i \rangle$  messages and replied with  $\text{PHASE1B}$  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  $\text{MATCHA}(i, C_i)$ , it did not contain a configuration for round  $j$  in its log. Moreover, by processing the  $\text{MATCHA}(i, C_i)$  request and inserting  $C_i$  in log entry  $i$ , the matchmaker is guaranteed to never process a  $\text{MATCHA}(j, C_j)$  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.  $x$  is the value proposed in round  $k$ , so no other value could be chosen in round  $k$ .

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

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

### 3.4 Optimizations

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

**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 (remember that the set of rounds can be *any* unbounded totally ordered set with a least element). 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:

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

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 that this optimization is essential for implementing Matchmaker MultiPaxos with 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, 33] 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.

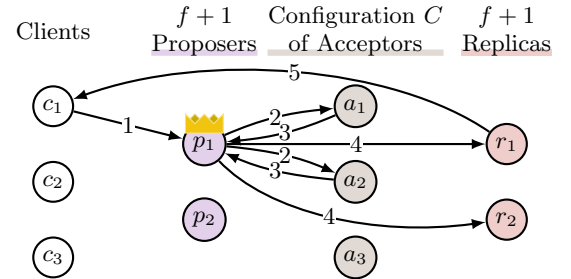


Figure 4: An example execution of MultiPaxos ( $f = 1$ ). The leader is adorned with a crown.

To agree on a log of commands, MultiPaxos implements one instance of Paxos for every log entry. The  $i$ th 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  $\text{PHASE1A}(i)$  message that acts as the  $\text{PHASE1A}$  message for every log entry larger than  $w$ . Also, an acceptor replies with a  $\text{PHASE1B}(i, vr, vv)$  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  $\text{PHASE1B}$  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.

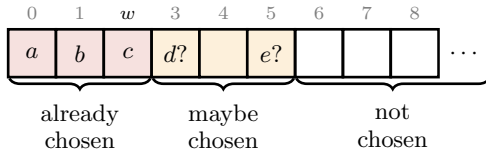


Figure 5: A leader’s knowledge of the log after Phase 1.

After Phase 1, the leader sends a  $\text{PHASE2A}$  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 performed only 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.

The leader then executes the Matchmaking phase in exactly the same way as in Matchmaker Paxos (i.e. it sends  $\text{MATCHA}(i, C_i)$  messages to the matchmakers and awaits  $\text{MATCHB}(i, H_i)$  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  $C_{\text{old}}$  in round  $i$  to some new configuration  $C_{\text{new}}$ , the leader of round  $i$  simply advances to round  $i + 1$  and selects the new configuration  $C_{\text{new}}$ . 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 executes Phase 1 only once per leader change (and not once per command), Matchmaker MultiPaxos runs the Matchmaking phase and Phase 1 only when a new leader is elected or when a leader changes its round (e.g., when a leader transitions from round  $i$  to round  $i + 1$  as part of a reconfiguration). In the normal case (i.e. during Phase 2), Matchmaker MultiPaxos and MultiPaxos are identical. Matchmaker MultiPaxos does not introduce *any* overheads in the normal case.

Further 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 used in some round  $j < i$ .

Finally note that, in theory, the Matchmaking phase of Matchmaker Paxos is not fully live (which is fundamentally unavoidable [6]). If multiple proposers simultaneously execute the Matchmaking phase, it is possible to experience “dueling proposers” [15]. Matchmaker MultiPaxos avoids

this in practice using leader election. A proposer only initiates the Matchmaking phase if it is elected leader.

#### 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  $i$ .

**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 scheme 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 discuss Matchmaker Paxos, then 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. See 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

1: **upon** receiving  $GARBAGEA\langle i \rangle$  from proposer  $p$  **do**

2:   **for all**  $j < i$  **do**

3:     delete  $L[j]$

4:    $w \leftarrow \max(w, i)$

5:   send  $GARBAGEB\langle i \rangle$  to  $p$

6: **upon** receiving  $MATCHA\langle i, C_i \rangle$  from proposer  $p$  **do**

7:   **if**  $i < w$  or  $\exists C_j$  in round  $j \geq i$  in  $L$  **then**

8:     ignore the  $MATCHA\langle i, C_i \rangle$  message

9:   **else**

10:      $H_i \leftarrow \{(j, C_j) \mid C_j \in L\}$

11:      $L[i] \leftarrow C_i$

12:     send  $MATCHB\langle i, w, H_i \rangle$  to  $p$

---

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

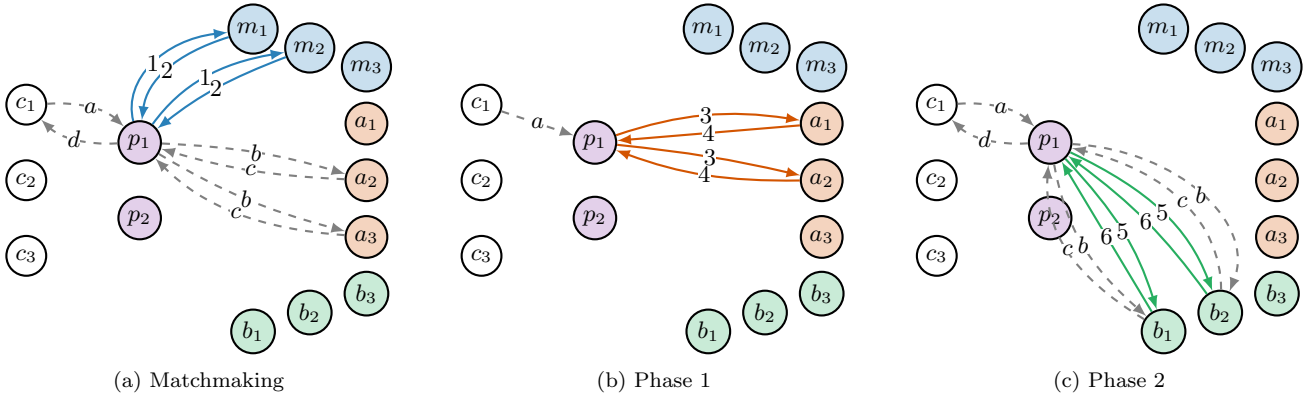


Figure 6: An example Matchmaker MultiPaxos reconfiguration without Optimization 2. The leader  $p_1$  reconfigures from the acceptors  $a_1, a_2, a_3$  to the acceptors  $b_1, b_2, b_3$ . Client commands are drawn as gray dashed lines. Note that every subfigure shows one phase of a reconfiguration using solid colored lines, but the dashed lines show the complete execution of a client request that runs concurrently with the reconfiguration. For simplicity, we assume that every proposer also serves as a replica.

Once a proposer receives  $\text{GARBAGEB}\langle i \rangle$  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  $\text{GARBAGEA}\langle i \rangle$  command. We explain the three situations and provide intuition on why they are safe. We refer to the technical report for a proof [35].

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

**Scenario 2.** If the proposer  $p_i$  executes Phase 1 in round  $i$  and finds  $k = -1$ , then it can safely issue a  $\text{GARBAGEA}\langle i \rangle$  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  $\text{GARBAGEA}\langle i \rangle$  command after it informs a Phase 2 quorum of acceptors in  $C_i$  of this fact. Any future proposer  $p_j$  in round  $j > i$  will contact a Phase 1 quorum of  $C_i$  and encounter at least one acceptor that knows the value  $x$  has already been chosen. When this acceptor informs  $p_j$  that a value  $x$  has already been chosen,  $p_j$  stops executing the protocol entirely and simply fetches the value  $x$  from one of the  $f + 1$  machines that store the value. Note that the decision of exactly which  $f + 1$  machines is not important. Any  $f + 1$  machines can be used. In our implementation, we use  $f + 1$  replicas.

## 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  $\text{GARBAGEA}\langle i \rangle$  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 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 commands have been stored on  $f + 1$  replicas. It then issues a  $\text{GARBAGEA}\langle i \rangle$  command to the matchmakers and awaits



$f + 1$  GARBAGEB( $i$ ) 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

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

Reconfiguring proposers and replicas is straightforward. In fact, Matchmaker MultiPaxos reconfigures proposers and replicas in exactly the same way as MultiPaxos [33], 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 proposers perform the Matchmaking phase only during a change in round. Thus, for the vast majority of the time—specifically, when there is a single, stable leader—the matchmakers are idle. When idle, they don’t send or receive any messages and are 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  $M_{\text{old}}$  of matchmakers to a new set  $M_{\text{new}}$  as follows. First, a proposer (or any other node, it doesn’t really matter) sends a STOPA( $\rangle$ ) message to the matchmakers in  $M_{\text{old}}$ . When a matchmaker  $m_i$  receives a STOPA( $\rangle$ ) message, it stops processing messages (except for other STOPA( $\rangle$ ) messages) and replies with STOPB( $L_i, w_i$ ) 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  $w$ .

The proposer then sends  $L$  and  $w$  to all of the matchmakers in  $M_{\text{new}}$ . Each matchmaker adopts these values as its initial state. At this point, the matchmakers in  $M_{\text{new}}$  *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  $M_{\text{new}}$  and  $M'_{\text{new}}$ . To avoid this, every matchmaker in  $M_{\text{old}}$  doubles as a Paxos acceptor. A proposer attempting to reconfigure to  $M_{\text{new}}$  acts as a Paxos proposer and gets the value  $M_{\text{new}}$

chosen by the matchmakers (which are acting as Paxos acceptors). Once  $M_{\text{new}}$  is chosen, the proposer notifies the matchmakers in  $M_{\text{new}}$  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 [35].

## 7. INSIGHTS AND GENERALITY

### 7.1 Insights

**Vertical Paxos** Matchmaker Paxos was inspired by Vertical Paxos [17], but differs in ways that improve simplicity, efficiency, and garbage collection. 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 executes Phase 1 in order to perform a reconfiguration. Thus, Vertical Paxos cannot be extended to MultiPaxos without causing performance degradation during reconfiguration. This is not the case for matchmakers thanks to Phase 1 Bypassing. Finally, 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.

**Fast Paxos.** Fast Paxos [14] 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. 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 a fixed set of  $f + 1$  acceptors (and hence with  $f + 1$ -sized quorums). Specifically, we deploy Fast Paxos with  $f + 1$  acceptors, with a single unanimous Phase 2 quorum, and with singleton Phase 1 quorums. A full description of the protocol and a proof of correctness is given in our technical report [35].

**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. We also discovered that DPaxos’ garbage collection algorithm is unsafe. Matchmaker MultiPaxos’ garbage collection algorithm fixes the bug (see our technical report for details [35]).

### 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  $\alpha$  is some configurable parameter. Note that the MultiPaxos leader can process at most  $\alpha$  unchosen commands at a time.

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 [13, 23, 36, 2, 31, 37, 32].

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 [35], for example, is an EPaxos [23] variant that partially orders commands into a graph. BPaxos is a modular protocol that uses Paxos as a black box subroutine. Due to this modularity, we can directly replace Paxos with Matchmaker Paxos to support reconfiguration. The same idea can also be applied to EPaxos. CASPaxos [31] 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 makes Matchmaker Paxos an attractive foundation on top of which other non-log based protocols can build their own reconfiguration protocols.

## 8. EVALUATION

We now evaluate Matchmaker MultiPaxos. Refer to our technical report [35] for further evaluation. Our implementation of Matchmaker MultiPaxos is available at [github.com/mwhittaker/frankenpaxos](https://github.com/mwhittaker/frankenpaxos). Matchmaker MultiPaxos is implemented in Scala using the Netty networking library. We deployed Matchmaker MultiPaxos on m5.xlarge AWS EC2 instances within a single availability zone. 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. In particular, any two logical roles can be placed on the same machine, so long as the two roles are not the same. For example, we can co-locate a leader, an acceptor, a replica, and a matchmaker, but we can’t co-locate two acceptors (without reducing the fault tolerance of the system). 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.

### 8.1 Reconfiguration

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

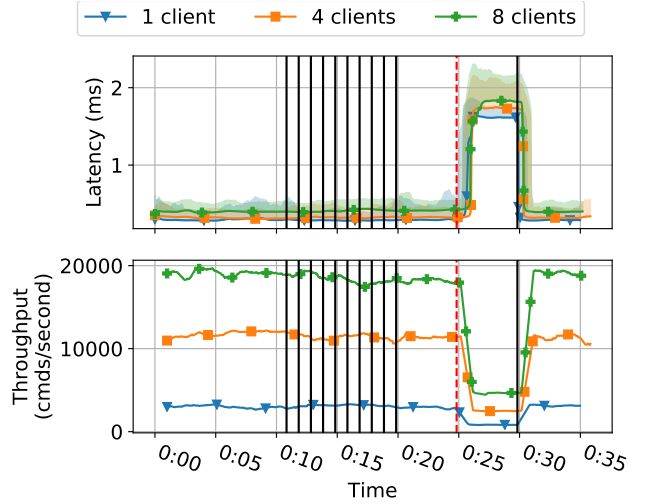


Figure 7: Matchmaker MultiPaxos’ latency and throughput ( $f = 1$ ). Median latency is shown using solid lines, while the 95% latency is shown as a shaded region above the median latency. The vertical black lines show reconfigurations. The vertical dashed red line shows an acceptor failure.

We also perform this experiment with an implementation of MultiPaxos with horizontal reconfiguration. We set  $\alpha$  to 8. Because  $\alpha$  is equal to the number of clients, MultiPaxos never stalls because of an insufficiently large  $\alpha$ .

**Results.** The latency and throughput of Matchmaker MultiPaxos for  $f = 1$  is shown in Figure 7. For space reasons, we report all of our results only for  $f = 1$ . The results for larger values of  $f$  are similar and shown in our technical report [35]. Note that Matchmaker MultiPaxos’ throughput decreases and its latency increases as we increase  $f$ . The overall trends, however, are the same. Throughput and latency are both computed using sliding one second windows. Median latency is shown using solid lines, while the 95% latency is shown as a shaded region above the median latency. The black vertical lines denote reconfigurations, and the red dashed vertical line denotes the acceptor failure.

The medians, interquartile ranges (IQR), and standard deviations of the latency and throughput (a) during the first 10 seconds and (b) between 10 and 20 seconds are shown in Table 1. For latency, reconfiguration has little to no impact (roughly 2% changes) on the medians, IQRs, or standard deviations. The one exception is that the 8 client standard deviation is significantly larger. This is due to a small number of outliers. Reconfiguration has little impact on median throughput, with all differences being statistically insignificant. The IQRs and standard deviations sometimes increase and sometimes decrease. The IQR is always less than 1% of the median throughput, and the standard deviation is always less than 4%.

For every reconfiguration, the new acceptors become active within a millisecond. The old acceptors are garbage collected within five milliseconds. This means that only one configuration is ever returned by the matchmakers. We implement Matchmaker MultiPaxos with an optimization called thriftiness [23]—where PHASE2A messages are sent to a randomly selected Phase 2 quorum—so the throughput and latency expectedly degrade after we fail an acceptor. Af-

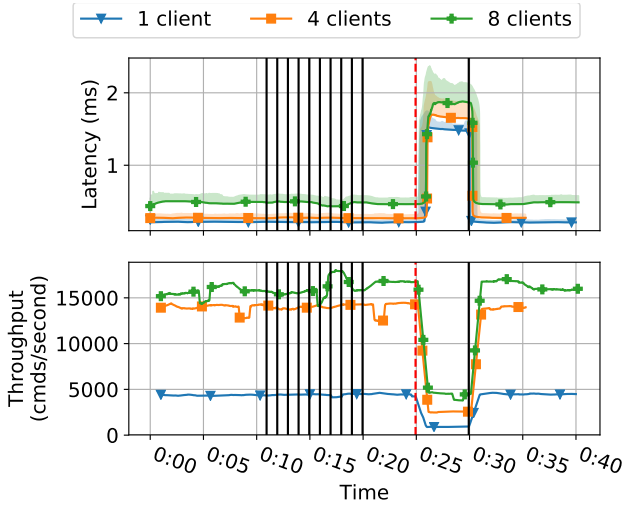


Figure 8: The latency and throughput of MultiPaxos with horizontal reconfiguration ( $f = 1$ ).

Table 1: Figure 7 median, interquartile range, and standard deviation of latency and throughput.

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

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

ter we replace the failed acceptor, throughput and latency return to normal within two seconds.

The latency and throughput of MultiPaxos for  $f = 1$  is shown in Figure 8. As with Matchmaker MultiPaxos, MultiPaxos can perform a horizontal reconfiguration without any performance degradation.

## 8.2 Ablation Study

**Experiment Description.** We now perform an ablation study to measure the effects of the three optimizations presented in Section 3. We deploy Matchmaker MultiPaxos as above. Each benchmark runs for 20 seconds with eight clients. The leader reconfigures the set of acceptors five times throughout the course of the benchmark. Every benchmark is run with a different subset of the optimizations in Section 3 enabled. When garbage collection (Optimization 3) is disabled, no garbage collection is performed. When Phase 1 Bypassing (Optimization 2) is disabled, the leader does not process commands during Phase 1 of a reconfiguration. When Proactive Matchmaking (Optimization 3) is disabled, the leader does not process commands during the

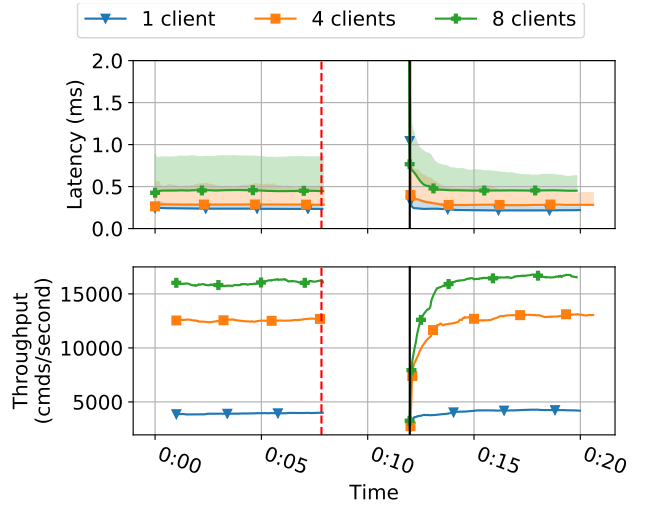


Figure 10: Matchmaker MultiPaxos' latency and throughput ( $f = 1$ ). The dashed red line denotes a leader failure.

Matchmaking phase of a reconfiguration. To simulate running on a wide area network—a common scenario [5, 1]—the acceptors and matchmakers delay their PHASE1B and MATCHB messages by 250 milliseconds. To keep normal case throughput and latency consistent with the previous benchmarks, PHASE2Bs are not delayed.

**Results.** The max latency (computed over 500 millisecond rolling windows) and throughput (250 millisecond windows) of Matchmaker MultiPaxos are shown in Figure 9. Without any optimizations, the latency of Matchmaker MultiPaxos expectedly spikes to 500 milliseconds after every reconfiguration, while the throughput drops to zero. Enabling garbage collection has little effect on throughput or latency, but garbage collection is required to shut down old configurations, so we cannot disable it in practice. With garbage collection and Phase 1 Bypassing, the latency peaks only at 250 milliseconds. The throughput drops to zero, but now only for 250 milliseconds. With all three optimizations enabled, the performance of the protocol is steady.

This ablation study serves three purposes. First, it highlights the importance of the three optimizations outlined in Section 3. Second, it highlights that Matchmaker MultiPaxos can mask the latency of a reconfiguration, even when the latency is large. Third, the ablation study is indirectly a comparison with stop-the-world reconfiguration protocols like the one used by Viewstamped Replication [19]. A stop-the-world reconfiguration protocol behaves like the unoptimized protocols in our ablation study. They stop processing commands entirely for some duration of time, so latency spikes and throughput drops to zero.

## 8.3 Leader Failure

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

**Results.** The latency and throughput of the benchmarks

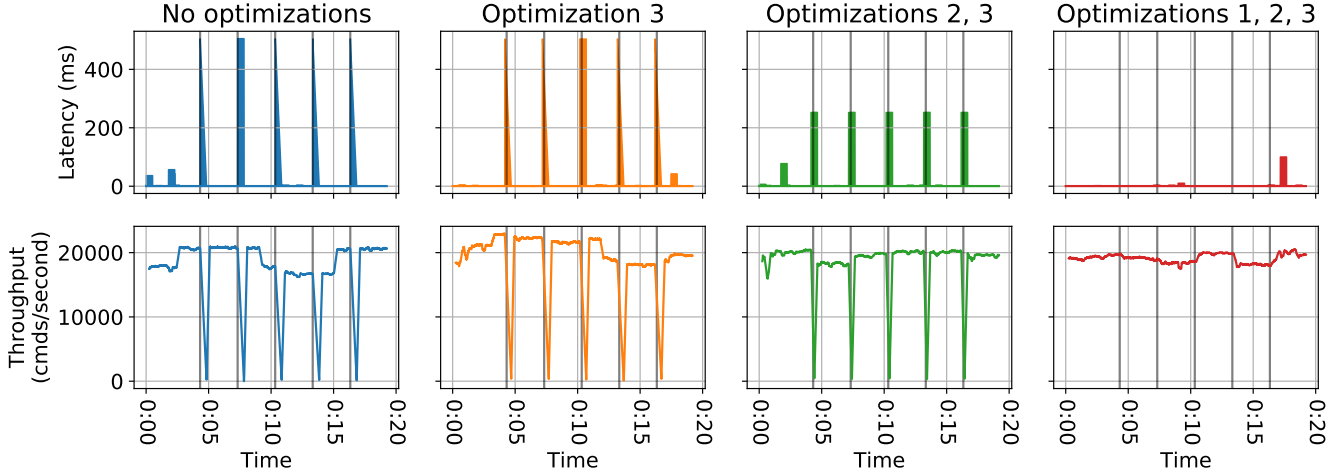


Figure 9: The latency and throughput of Matchmaker MultiPaxos with various optimizations enabled.

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. The throughput and latency return to normal within two seconds after a new leader is elected.

#### 8.4 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 remain 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.

### 9. RELATED WORK

**Raft.** Raft [28] 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

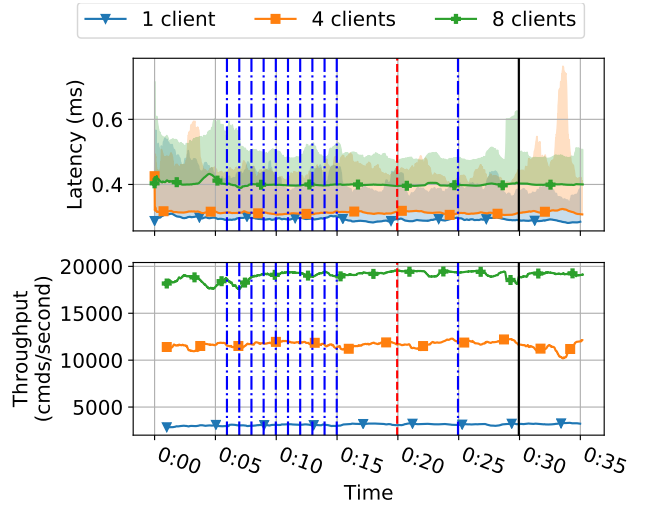


Figure 11: The latency and throughput of Matchmaker MultiPaxos ( $f = 1$ ). The dotted blue, dashed red, and vertical black lines show matchmaker reconfigurations, a matchmaker failure, and an acceptor reconfiguration respectively.

$i$  to skip Phase 1 in round  $i + 1$ , whereas with Phase 1 Bypassing, the leader instead uses Phase 1 information.

**ZooKeeper.** ZooKeeper, a distributed coordinated service, which uses ZooKeeper Atomic Broadcast [8] is a protocol similar to MultiPaxos that can also reconfigure quickly after leader failures.

### 10. CONCLUSION

We presented Matchmaker Paxos and Matchmaker MultiPaxos to address the lack of research on the increasingly important topic of reconfiguration. Our protocols 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] A. Ailijiang, A. Charapko, M. Demirbas, and T. Kosar. Wpaxos: Wide area network flexible consensus. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):211–223, 2019.
- [2] 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.
- [3] 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.
- [4] C. Chrysafis, B. Collins, S. Dugas, J. Dunkelberger, M. Ehsan, S. Gray, A. Grieser, O. Herrnsstadt, 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.
- [5] 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.
- [6] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.
- [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] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pages 245–256. IEEE, 2011.
- [9] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. Mdc: 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. Generalized consensus and paxos. *Technical Report MSR-TR-2005-33, Microsoft Research*, 2005.
- [14] L. Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.
- [15] L. Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.
- [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. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 312–313, 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. *MIT-CSAIL-TR-2012-021*, 2012.
- [20] 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.
- [21] 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.
- [22] 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.
- [23] 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.
- [24] 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.
- [25] 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.
- [26] D. Ongaro. Bug in single-server membership changes. <https://groups.google.com/forum/#!msg/raft-dev/t4xj6dJTP6E/d2D9LrWRza8J>. accessed: 2020-03-01.
- [27] D. Ongaro. *Consensus: Bridging theory and practice*. PhD thesis, Stanford University, 2014.
- [28] 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.
- [29] 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.
- [30] 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.
- [31] D. Rystsov. Caspaxos: Replicated state machines without logs. *arXiv preprint arXiv:1802.07000*, 2018.
- [32] 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.
- [33] R. Van Renesse and D. Altinbuken. Paxos made moderately complex. *ACM Computing Surveys (CSUR)*, 47(3):1–36, 2015.
  - [34] M. Vukolić et al. The origin of quorum systems. *Bulletin of EATCS*, 2(101), 2013.
  - [35] 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](https://mwhittaker.github.io/publications/matchmaker_paxos_tr.pdf).
  - [36] 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](https://mwhittaker.github.io/publications/compartmentalized_bipartisan_paxos.pdf).
  - [37] 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.