Rabia: Simplifying State-Machine Replication Through Randomization

Haochen Pan*
University of Chicago
Chicago, IL, USA
haochenpan@uchicago.edu

Jesse Tuglu*
University of Michigan
Ann Arbor, MI, USA
tuglu@umich.edu

Neo Zhou
Boston College
Boston, MA, USA
neo.zhou@bc.edu

Tianshu Wang*
Duke University
Durham, NC, USA
tw295@duke.edu

Yicheng Shen
Boston College
Boston, MA, USA
yicheng.shen@bc.edu

Xiong Zheng†
Google, Inc.
Kirkland, WA, USA
xiongzheng@google.com

Joseph Tassarotti
Boston College
Boston, MA, USA
tassarot@bc.edu

Lewis Tseng
Boston College
Boston, MA, USA
lewis.tseng@bc.edu

Roberto Palmieri
Lehigh University
Bethlehem, PA, USA
palmieri@lehigh.edu

Abstract

We introduce Rabia, a simple and high performance framework for implementing state-machine replication (SMR) within a datacenter. The main innovation of Rabia is in using *randomization* to simplify the design. Rabia provides the following two features: (i) It does not need any fail-over protocol and supports trivial auxiliary protocols like log compaction, snapshotting, and reconfiguration, components that are often considered the most challenging when developing SMR systems; and (ii) It provides high performance, up to 1.5x higher throughput than the closest competitor (i.e., EPaxos) in a favorable setup (same availability zone with three replicas) and is comparable with a larger number of replicas or when deployed in multiple availability zones.

CCS Concepts: • **Computer systems organization → Dependable and fault-tolerant systems and networks**; • **Computing methodologies → Distributed algorithms**.

*Work was done at Boston College.
†Work was done at UTexas Austin.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SOSP '21, October 26–29, 2021, Virtual Event, Germany
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8709-5/21/10…$15.00
https://doi.org/10.1145/3477132.3483582

Keywords: SMR, Consensus, Formal Verification

ACM Reference Format:
Haochen Pan, Jesse Tuglu, Neo Zhou, Tianshu Wang, Yicheng Shen, Xiong Zheng, Joseph Tassarotti, Lewis Tseng, and Roberto Palmieri. 2021. Rabia: Simplifying State-Machine Replication Through Randomization. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21), October 26–29, 2021, Virtual Event, Germany*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3477132.3483582

1 Introduction

State-Machine Replication (SMR) uses replication to ensure that a service is available and consistent in the presence of failures. One popular mechanism for implementing SMR is to use a consensus algorithm to agree on the total order of client requests (or commands), namely, a log-based SMR approach [6, 52, 60]. Paxos and variants [38, 39] had mostly been the de facto choice for implementing SMR, e.g., Chubby [16], Google Spanner [22], Microsoft Azure Storage [19]. Raft [52] recently became a popular alternative, designed with understandability as the priority. Many modern production systems choose Raft over Paxos, e.g., Redis [2], RethinkDB [4], CockroachDB [3], and etcd [1]. In this work, we aim to address one of the remaining challenges in building a *high-performance log-based SMR* system within a single datacenter: reducing the engineering effort and simplifying the integration of auxiliary protocols.

Paxos and variants are notoriously difficult to understand and integrate with SMR [20, 52]. Raft is sometimes considered easier to comprehend with its use of a stronger notion of leader; however, it still requires a tremendous amount of effort to develop a fully functional Raft-based SMR system [8, 9, 32]. One root cause is that prior practical consensus algorithms, including Paxos, Raft, and recent systems, require

auxiliary protocols (e.g., leader election, snapshotting, fail-over/recovery mechanism, log compaction/truncation) for ensuring high performance and liveness. We will elaborate on these challenges in Section 2.

Rabia is a *simple* SMR framework that ensures a total order of client requests and achieves high performance in a single datacenter setting. Rabia does *not* need any fail-over protocol and supports trivial log compaction, reconfiguration, and snapshotting protocols. We accomplish this through a novel implementation of consensus that leverages a RAndomized BInary Agreement protocol.[1]

Randomized algorithms are known to have less stable performance compared to deterministic ones. By limiting our focus to SMR systems deployed in a datacenter where the network infrastructure is *stable*, we can achieve higher throughput than SMR systems implementing total order through Multi-Paxos [38, 39], and Egalitarian Paxos (EPaxos) [48] in a favorable setup (same availability zone with three replicas), and have comparable performance when deployed with a larger number of replicas or in multiple zones.

We also present RedisRabia, an integration of Rabia and Redis, a popular distributed in-memory data store. We compare RedisRabia with (i) Redis (synchronous) replication that is used widely in production, and (ii) RedisRaft [2], which is Redis integrated with Raft. RedisRabia has comparable throughput with synchronous replication with the same fault-tolerance, and outperforms RedisRaft by 2.5x with batching.

Challenges and Key Observation: Randomized consensus algorithms are known to have a simpler structure than deterministic consensus algorithms [11, 13]; however, there are two major challenges in deploying them in practical systems. First, most randomized algorithms have less stable performance due to randomized rules that depend on the outcome of a "coin flip." That is, the performance is probabilistic in nature. Second, the latency of randomized algorithms is sub-optimal. All randomized consensus algorithms in the literature have at least four message delays *on average*, whereas prior deterministic consensus algorithms, e.g., [38, 39, 52], have optimal fast-path latency of two message delays and stay on the fast path when there is a stable leader.

We observe that it is possible to make a randomized *binary* consensus algorithm fast in current network infrastructures. In fact, Rabia achieves the best performance if a majority of replicas receive a similar set of messages, which is typical in modern networks. Rabia enables its underlying consensus algorithm to reach agreement in three message delays on the fast path. According to our tests using public commercial clouds and private clusters, current network infrastructures easily provide these characteristics (we call these networks *stable*). This observation holds even with legacy network hardware (e.g., without recent RDMA cards). We also empirically show Rabia suffers only a minor performance loss when deployed in multiple availability zones in Google Cloud Platform. This implies that the network conditions that allow Rabia to reach consensus fast are practical, not stringent.

However, a binary consensus algorithm can only take a binary input (0 or 1) and generate a binary output. The log-based SMR design requires a multi-valued consensus algorithm so that the replicas can agree on the order of client requests. Therefore, to make use of fast randomized binary consensus, the next technical challenge Rabia addresses is to efficiently convert binary consensus into a (weaker) form of multi-valued consensus that can be used for SMR.

Key Techniques: In Rabia, a client sends requests to an assigned replica, which then relays them to all the other replicas. Each replica stores pending requests that are not added to the log yet in its local *min* priority queue (PQ). The key to the min PQ is the request timestamp. This design allows the replicas to have the same head of its PQ most of the time, even under high workload, in a stable network. Replicas then use our new consensus algorithm, Weak-MVC, to agree on the request for each slot of the log.

Weak-MVC is a novel implementation of a *relaxed version* of multi-valued consensus. Each replica's input to Weak-MVC is the *oldest pending request* in its local PQ. The choice of using the oldest request as the input to our algorithm allows us to achieve high performance in a stable network. If most replicas propose the same request, then Weak-MVC terminates in three message delays. In our evaluation, stable networks allow Rabia to use the fast path 99.58% of the time under all open-loop experiments.

Replicas may propose different requests for the same slot. The approach to address this scenario is our second novel design – *forfeiting a slot* – which is different from prior consensus algorithms, e.g., [26, 29, 38, 39, 48, 52, 67]. Instead of deciding *which* request to agree upon, we use binary consensus to determine *whether* there is an agreement. If not, we choose to "forfeit fast," i.e., allow a slot to store a NULL value. In this case, replicas can terminate (with a forfeited slot) also in three message delays.

The rationale behind our design is that, in a stable network that delivers the oldest pending request req_{old} to all replicas in a short time, replicas can soon reach an agreement on req_{old} in a future slot, even if replicas forfeit current slot(s). Furthermore, a stable and reliable network also avoids continuous forfeits because eventually req_{old} will be delivered. Compared to spending more communication to agree on a request, it is faster to forfeit the current slot, and move on to the next slot.

Highlights of Rabia: Rabia has the following features designed to favor widespread adoption and integration:

- **No Fail-over and Simple Log Compaction**: One major challenge in implementing an SMR system is

[1] These algorithms are also called agreement algorithms, but we will refer to them as consensus algorithms.

the integration of the auxiliary protocols, including fail-over, reconfiguration, snapshotting, and log compaction protocol (a garbage collection mechanism that discards older log slots from memory) [8, 9, 20, 32, 33]. Prior consensus algorithms, including both leader-based designs (e.g., Paxos [38, 39] and Raft [52]) and multi-leader-based designs (e.g., Mencius [43], EPaxos [48], M^{2}Paxos [56], Caesar [10], and Atlas [25]), require a complicated fail-over mechanism to recover from the failure of a leader or a command leader, which also makes other auxiliary protocols challenging.
In contrast, Rabia does not need a fail-over. Intuitively, this is because Rabia uses randomized consensus instead of deterministic consensus to agree on the ordering of client requests, which ensures that at all times, non-faulty replicas are guaranteed to be able to learn the decision. This guarantee also enables a trivial log compaction mechanism in Rabia (which can be specified in three lines of pseudo-code), and simple reconfiguration and snapshotting protocols, as presented in Section 4.

- **Machine-checked proof**: In Section 5, we briefly describe how we use the Ivy [46] and Coq [62] theorem provers to formally verify the safety of our protocol. While a number of protocols and distributed systems have been verified, including Raft and many variants of Paxos [34, 53, 61, 64, 66], we are not aware of any verification of the multi-leader-based designs described above that are Rabia's closest competitors. The full proof is presented in our technical report [54].
- **High performance**: We implement Rabia in Go, and compare against Multi-Paxos and EPaxos in Google Cloud Platform, where the network is stable. In our setting, EPaxos with no conflicting workload is the best competitor. When deployed in the same availability zone with three replicas, Rabia's maximum speedup in throughput is up to 1.5x using a close-loop test, despite its quadratic communication complexity and sub-optimal latency on the fast path. In other cases, Rabia has comparable performance despite its higher message complexity.

Outline: We first enumerate the challenges of implementing existing consensus algorithms and discuss the background in Section 2. We then present the complete framework of Rabia and its analysis in Section 3. Practical considerations are discussed in Section 4. The proofs of our safety and liveness properties appear in Section 5. We evaluate Rabia in Section 6. Related work is presented in Section 7.

2 Motivation and Background

Why Another Simple Consensus Algorithm? The first barrier to implementing a consensus algorithm is understandability [20, 33, 52, 63]. The intuition behind Paxos and variants is difficult to grasp. Such a challenge was perfectly summarized by the developer of Google's Chubby system [20] (which is based on Multi-Paxos): *"There are significant gaps between the description of the Paxos algorithm and the needs of a real-world system . . . the final system will be based on an unproven protocol."* Most recent consensus algorithms proposed are inspired by Paxos, e.g., [25, 44, 48, 56], and share a similar challenge in implementation.

Raft [52] addresses the understandability issue by using a stronger notion of leader to simplify the conceptual design. However, developing a Raft-based SMR system is still quite challenging, even for seasoned developers, e.g., [8, 9, 32, 33]. It is because of the second barrier, namely *engineering complexity*.

While the authors of Raft provided a comprehensive overview of the necessary auxiliary protocols [32, 33], it is still difficult to develop and integrate them with the core consensus algorithm. For example, Redis [9] and RethinkDB [4] had bugs in their preliminary integration of Raft into their respective systems. In fact, even after Jepsen [9] identified 21 major bugs back in June 2020, RedisRaft [2] showed several performance bugs in our experience (e.g., frequent leader changes when there is no failure, nor message drop). Industry blog posts [8, 9, 33] identified fail-over, log compaction, reconfiguration, and snapshotting as major concerns when developing a Raft-based SMR system.

By using a randomized consensus as a core component, Rabia does not need a fail-over protocol, and hence supports trivial log compaction, snapshotting, and reconfiguration. In addition, Rabia uses two novel designs to achieve high performance in a stable network. We only found two competitors that achieve a similar goal: single-decree Paxos [38][2] and NOPaxos (Network-Ordered Paxos) [41]. Section 7 discusses recent Byzantine fault-tolerance systems (e.g., HoneyBadger [47] and Algorand [21, 31]) that also utilize randomized consensus algorithms.

Single-decree Paxos does not need a fail-over, because any replica can help others recover an agreed value in any slot. However, it suffers from performance loss, compared to Multi-Paxos, and in general, has unstable performance. In fact, there is no bounded analysis on the average delay. Rabia is proven to have five message delays on average, as analyzed in Section 5. NOPaxos uses the network fabric to sequence requests (i.e., a sequencer in switches) to simplify design and improve performance. As a result, NOPaxos outperforms Multi-Paxos by 4.7x in throughput [41]. However, its fail-over and reconfiguration require switches to install new forwarding rules, which, in our opinion, make the development more complicated. In addition, most public clouds

[2]Single-decree Paxos is designed to reach an agreement on a single slot, whereas Multi-Paxos skips Phase 1 to achieve a higher performance with a stable leader in the case of multiple slots.

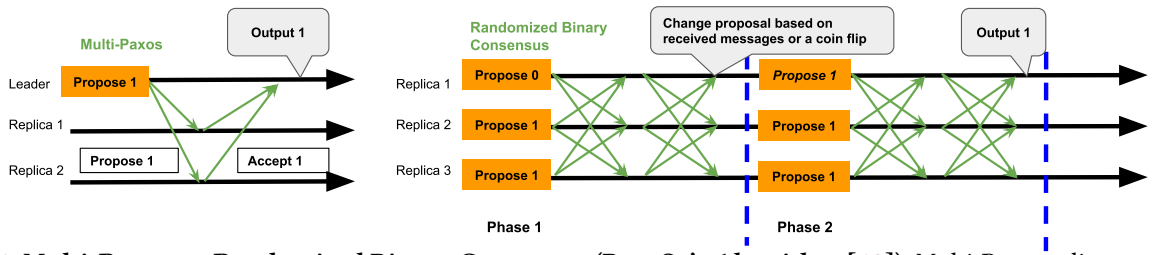


Figure 1. Multi-Paxos vs. Randomized Binary Consensus (Ben-Or's Algorithm [13]). Multi-Paxos relies on a leader to make a decision (for a slot), whereas Ben-Or's algorithm requires replicas to make a "joint decision." In the common case, Multi-Paxos terminates in 2 message delays, whereas a randomized consensus algorithm might take multiple phases. In this particular example, Ben-Or's algorithm takes 2 phases (and 4 message delays) to terminate.

do not allow users to directly implement their own logic in switches, which makes NOPaxos less adoptable in general.

SMR: State-Machine Replication (SMR) ensures that a service is available and consistent in the presence of failures through replication. SMR ensures linearizability, or strict serializability in case of transactions. We adopt the log-based SMR approach [6, 52, 60], which works as follows: (i) each client submits a request to replicas (or servers) and waits for a response; (ii) each replica has a log that stores client requests; (iii) the log is divided into slots; (iv) replicas use a consensus protocol to agree on the ordering of the client requests, i.e., the request to be stored in each slot of the log; (v) replicas apply the requests (i.e., execute the operations in the request) slot by slot; and (vi) after applying the request, a replica sends a response to the corresponding client.

Following prior works (e.g., [6, 48, 52]), Rabia stores the log in main memory. This design is reliable as long as the number of simultaneous replica failures is bounded. For deterministic services, all the replicas have the same state after applying the request in a given slot, because at that point, replicas have executed the same set of operations in the same order.

Consensus: The core component of the log-based SMR design is its consensus algorithm (step (iv) above). A consensus protocol is correct if it ensures safety and liveness properties. For each slot of the log, liveness (or *termination*) ensures that all the non-faulty replicas eventually obtain a request. Also, for each slot, safety states two conditions: (*agreement*) each non-faulty replica obtains the same request; and (*validity*) each obtained request was proposed by a client.

The famous FLP result [27] implies that deterministic consensus algorithms work only in partially synchronous systems. For example, Paxos (and its variants) and Raft ensure safety at all times, but ensures liveness only when the leader is stable and can reach a quorum of replicas.

Randomized Consensus: Another approach to circumvent the FLP result is through randomization. Ben-Or [13] proposed a simple randomized binary consensus algorithm in PODC 1983, which achieves the following *probabilistic termination* property: each non-faulty replica eventually obtains a request *with probability 1* – as time proceeds, the probability for a replica to terminate approaches 1. The reason for the probabilistic termination is that it breaks a tie by flipping a coin [13]. Hence, an instance of this consensus protocol may take an arbitrarily long time to terminate. In fact, the average latency of Ben-Or's original algorithm is exponential with respect to the number of replicas, because each replica flips its own local coin. The latency is measured as the message delay from one replica to the other.

Figure 1 compares the difference between Multi-Paxos and Ben-Or's algorithm. The left part illustrates Paxos (with a stable leader). Upon receiving an acknowledgment from another replica, the leader can safely output the value it proposed, value "1" in this example. In comparison, Ben-Or's design does not rely on the notion of a leader. Instead, all the replicas proceed in phases, in which they need to propose and make a "joint decision." Replicas may propose different values. In this case, replicas use a randomized rule (i.e., the outcome of a coin flip) to break a tie. In the example, Replica 1 changes from "0" to "1" in Phase 2. Then, all the replicas terminate and output 1 in this phase. In Section 3.4, we discuss in more detail why such a "joint decision" design does not need a fail-over protocol, and hence simplifying the complexity in developing and integrating auxiliary protocols.

Challenges: Compared to highly optimized deterministic consensus algorithms like Paxos and Raft, randomized algorithms have a few limitations. As illustrated in Figure 1, the latency of a randomized consensus algorithm is less predictable, because the number of rounds may depend on the outcome of a coin flip. Moreover, the best-case latency is also worse than Multi-Paxos. Concretely, we identify two major challenges in designing a high-performance SMR system based on a randomized consensus algorithm: (i) how to stay on the fast path as often as possible? and (ii) how to reduce tail latency? We discuss how Rabia addresses these challenges in Section 3.2.

3 The Rabia SMR Framework

We consider an asynchronous distributed system consisting of n replicas (or servers). At most f of the replicas may fail by crashing, i.e., a fail-stop model. Rabia ensures safety and liveness as long as $n \geq 2f + 1$. We do not consider Byzantine

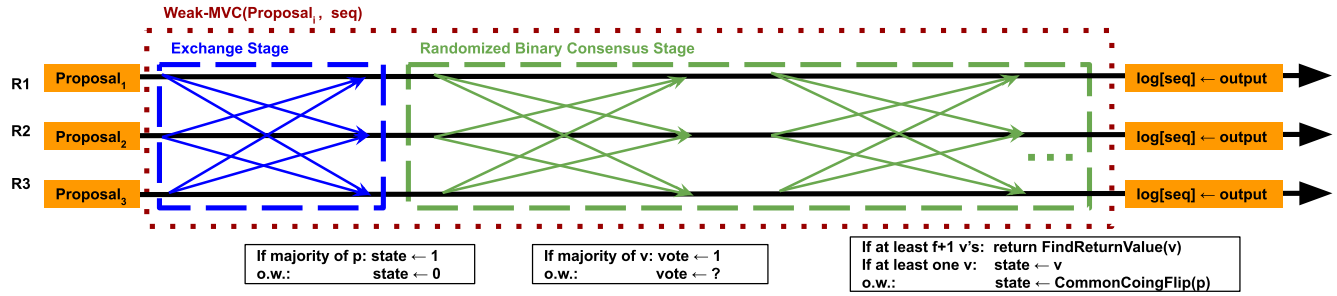


Figure 2. Execution Flow of Rabia. The core component is our consensus algorithm – Weak-MVC – whose communication pattern of the fast path is presented inside the red dotted box. The rules to update internal states are also listed underneath.

faults. The set of replicas is assumed to be static. An approach for reconfiguring the set of replicas is presented in Section 4.

3.1 Execution Flow

Rabia adopts the log-based SMR design [60]. Each replica has a log, and the goal is to agree on a client request (or a batch of requests) to be stored in each slot of the log. We present the pseudo-code of each replica N_i in Algorithm 1. Figure 2 illustrates the flow and rules for Rabia to agree on the request for a particular slot when $n = 3$.

Algorithm 1 Rabia: Code for Replica N_i

Local Variables:
- PQ_i ▷priority queue, initially empty
- seq ▷current slot index, initially 0

Code for Replica N_i:
1: **while** *true* **do**
2: $proposal_i \leftarrow$ first element in PQ_i that is not already in log ▷$proposal_i = i$'s input to Weak-MVC
3: $output \leftarrow$ Weak-MVC($proposal_i, seq$)
4: $log[seq] \leftarrow output$ ▷Add $output$ to current slot
5: **if** $output = \perp$ **or** $output \neq proposal_i$ **then**
6: $PQ_i.push(proposal_i)$
7: $seq \leftarrow seq + 1$

/* Event handler: executing in background */
Upon receiving ⟨Request, c⟩ **from client** c:
8: $PQ_i.push(\langle \text{Request}, c \rangle)$
9: forward ⟨Request, c⟩ to all other replicas

/* Executing in background periodically */
Log Compaction:
10: **for** each j-th slot in the log **do**
11: **if** $log[j]$ has been executed locally **then**
12: truncate $log[j]$ ▷Discard it or take a snapshot

A client sends a request to the assigned replica (a client proxy) and waits for a response from the same replica. Upon receiving the client request (Line 8 and 9), N_i first pushes the request to a *min priority queue* PQ_i and then forwards it to all other replicas. N_i uses PQ_i to store pending client requests, i.e., requests that have not been stored in the log yet. The head of the priority queue stores the *oldest pending request*. That is, N_i uses the request timestamp as the key for PQ_i. For brevity, the timestamp is omitted in Algorithm 1.

Replicas use a while loop to continuously agree on the request for each slot seq as long as their PQ is non-empty (Line 1 to 7). All the replicas participate in our consensus algorithm, Weak-MVC (Weak Multi-Valued Consensus), to agree on a request for slot seq (Line 2 and 3). N_i's input to the Weak-MVC consists of a request extracted from PQ_i, and the current slot index seq. Due to concurrency and message delay, PQ_i may contain some request r that is actually already stored in the log, because replicas have already agreed to store r in a prior slot. N_i discards such requests, and moves to the next request in PQ_i. The condition at Line 2 can be checked efficiently using a dictionary. Section 4 discusses why this dictionary will not grow unbounded.

Once Weak-MVC terminates, the output is written into the current slot (Line 4). Weak-MVC is designed in a way that N_i obtains an output with probability 1 as long as a majority of replicas, including i, are alive. The output of Weak-MVC may be a proposal from some replica or a NULL value $\perp$. A key design choice is to allow a slot to store a NULL value. In this case, N_i learns that the slot is "forfeited" and has to push its proposal back into PQ_i for a future slot (Line 6). Figure 2 illustrates the flow of agreeing on a request for the slot seq. The red dotted box presents an example execution of Weak-MVC, whose design is presented in Section 3.3.

Unlike prior SMR systems, e.g., [38, 39, 48, 52], performing log compaction in Rabia is very simple because the agreement and liveness properties of the Weak-MVC protocol ensure that all the non-faulty replicas will eventually agree on the same request or NULL for each slot. Therefore, each replica can discard slot j or take a snapshot once it has executed the request stored in the slot (Line 10 to 12). In prior systems, the log compaction mechanism needs to be carefully integrated with a leader election or a fail-over protocol to ensure that the discarded log can be safely recovered if failures occur. Such integration is not always straightforward [20, 25, 52].

3.2 Addressing performance challenges of randomized consensus

Before delving into details of Weak-MVC, let us first investigate how our framework presented in Algorithm 1 addresses the challenges mentioned in Section 2: (i) staying on the fast path as often as possible and (ii) avoiding long tail latency in a stable network. As it will become clear later, Weak-MVC is guaranteed to use the fast path (which *terminates in three message delays*), if

- (i) all replicas have the same proposal; or
- (ii) no majority of replicas propose the same request.

While these two conditions might not always occur, in our experience, they were often satisfied using a stable network.

Condition (i) together with using the *oldest pending request* (stored as the head of PQ_i) as a proposal allows the agreement to be made using mostly the fast path in a stable network. This is based on the two following observations: first, each replica forwards a request to other replicas when it receives an incoming request (Line 9 of Algorithm 1); and second, if message delay is small compared to the interval between two consecutive requests, then it is highly likely that most replicas have the same oldest pending request r in its local priority queue shortly after that request r was sent.

Condition (ii) reduces the chance for a long tail latency. In most randomized binary consensus algorithms (e.g., [13, 26, 29, 49]), the slow case occurs when roughly half of the replicas propose 0 and the others propose 1. In an unlucky sequence of message delays and coin flips, it might take a long time to terminate because replicas do not observe enough concordant values to jointly make a decision. Therefore, we design Weak-MVC in a way that if it seems difficult to terminate fast, then replicas choose to forfeit the proposal, and Weak-MVC outputs a NULL value $\perp$ in this case.

In our experience, it is much more efficient to "forfeit a slot" than to agree on some client request. Recall that the cause behind a forfeit is that replicas had extracted different proposals. During the time that replicas forfeit a slot, the oldest proposal that has not been agreed upon is highly likely to be propagated to all the replicas in a stable network. These replicas would then store this proposal in local PQs, and Rabia can hit the fast path again on the next slot after forfeiting. Therefore, our forfeit-fast approach is more efficient than agreeing on a non-NULL value at all times. The latter approach [26, 29, 67] takes logarithmic time with respect to the size of the client requests. Section 4 presents two other practical solutions that further improve the fast-path latency and reduce tail latency.

3.3 Weak-MVC

Allowing a slot to forfeit and contain a NULL value, Rabia violates the validity property by typical consensus algorithms (i.e., an output has to be a client request). Instead, it achieves the following relaxed version of validity:

Algorithm 2 Weak-MVC: Code for Replica i

When Weak-MVC is invoked with input q and seq:

```
1:  // Exchange Stage: exchange proposals
2:  Send (Proposal, q) to all                          ▷q is client request
3:  wait until receiving ≥ n − f Proposal messages
4:  if request q appears ≥ ⌊n/2⌋ + 1 times in Proposals then
5:      state ← 1
6:  else
7:      state ← 0
8:  // Randomized Binary Consensus Stage (Phase p ≥ 1)
9:  p ← 1                                              ▷Start with Phase 1
10: while true do
11:     /* Round 1 */
12:     Send (State, p, state) to all                  ▷state can be 0 or 1
13:     wait until receiving ≥ n − f phase-p State messages
14:     if value v appears ≥ ⌊n/2⌋ + 1 times in States then
15:         vote ← v
16:     else
17:         vote ←?
18:     /* Round 2 */
19:     Send (vote, p, vote) to all                    ▷vote can be 0, 1 or ?
20:     wait until receiving ≥ n − f phase-p vote messages
21:     if a non-? value v appears ≥ f + 1 times in votes then
22:         Return FindReturnValue(v)                  ▷Termination
23:     else if a non-? value v appears at least once in votes then
24:         state ← v
25:     else
26:         state ← CommonCoinFlip(p)                  ▷p-th coin flip
27:     p ← p + 1                                      ▷Proceed to next phase
```

Algorithm 3 Weak-MVC: Helper Function

Procedure FindReturnValue(v)

```
1: if v = 1 then
2:     Find value m that appears ≥ ⌊n/2⌋ + 1 times
       in Proposal messages received in Phase 0
3:     Return m
4: else
5:     Return ⊥                                        ▷return null value
```

> **Weak Validity**: the value stored in each slot of the log must either be a request from some client or a NULL value $\perp$.

Our algorithm, Weak-MVC (Weak Multi-Valued Consensus), achieves agreement, weak validity, and probabilistic termination as long as a majority of replicas are correct. The pseudo-code is presented in Algorithm 2 and 3.

Pseudo-code and Illustration: Weak-MVC has two stages: exchange stage (Line 1 to 7 in Algorithm 2) and randomized binary consensus stage (Line 8 to 27 in Algorithm 2). The first stage takes one phase (one message delay). The second stage takes one phase (two message delays) on the fast path and may take more phases in an unlucky sequence of message delays and coin flips.

The second stage is an adaption of Ben-Or's algorithm [13]. Intuitively, we replace the per-replica local coins used by

Ben-Or with a common coin (Line 26 of Algorithm 2), whose value is the same across all replicas in the same phase. Section 4 briefly discusses an efficient implementation of a common coin by having all replicas use a pseudo-random number generator or a hash with a common seed [58]. The seed can be configured at deployment, so there is no communication involved when using the common coin.

The red dotted box in Figure 2 presents the flow and the rules of Weak-MVC. The exchange stage and randomized binary consensus stage are shown in the blue and green dotted box, respectively. For brevity, we present the case when the randomized binary consensus stage terminates in one phase. The three green dots at the end of the red box denote the possibility that this stage might need more phases to agree on an output. We will prove later in Section 5 that Weak-MVC terminates with probability 1, and its average latency is five message delays.

Algorithm Description: Weak-MVC uses three types of messages: (i) Proposal message that carries a proposed value in the exchange stage (the input to Weak-MVC); (ii) State message that contains the state value, denoted by the variable *state* in the pseudo-code; and (iii) vote message that is used to cast votes in each phase. The value in the Proposal message is a client request. The state value is either 0 or 1, whereas the vote could take the value of 0, 1 or ?. The ? is a unique symbol denoting that the replica does *not* vote for 0 or 1, i.e., it gives up the vote in this phase. For messages sent in phase p, we will refer to it as phase-p messages. Each message is tagged with slot index *seq*, phase index p, and sender ID. For brevity, we omit *seq* and ID in the pseudo-code.

In the exchange stage, replicas exchange their proposals (a client request) at Line 2 of Algorithm 2, and decide the input to the second stage by updating the *state* variable based on the received proposals. In all communication steps, replicas wait for $n-f$ messages of a certain type (Line 3, 13, and 20 of Algorithm 2). Thus, Weak-MVC is *non-blocking*. Each replica sets *state* to 1 if it sees a majority of messages with the same request q (Line 5); otherwise, it sets *state* to 0 (Line 7). Using the majority as the threshold ensures that two replicas with $state = 1$ at the end of the exchange stage must have seen the same proposal from a majority of replicas.

The second stage, the randomized binary consensus stage, proceeds in phases, and each phase has two rounds. In the first round of phase p, replicas exchange their *state* variables and decide their vote for this phase. Each replica sets *vote* to v if they have seen a majority of phase-p State messages containing v, which is either 0 or 1; otherwise, the replica updates *vote* to ?. Using a typical quorum intersection argument on *state* messages, it is guaranteed that there is at most one non-? value in all the votes. That is, in the same round, we have either $vote \in \{0, ?\}$ or $vote \in \{1, ?\}$ at all the replicas. The proof of the safety property relies on this observation.

In the second round of phase p, replicas exchange their votes stored in the *vote* variable. If a non-? value v appears at least $f+1$ times in the received vote messages, then replicas can safely output a value using the helper function (Line 22). If a non-? value v appears at least once, then the replica sets *state* for the next phase as v (Line 24). If all phase-p vote messages received contain ?, then the replica flips a *common coin* to determine the state for the next phase (Line 26) – this is the randomized rule in Weak-MVC. CommonCoinFlip(p) denotes the p-th coin flip at each replica. A common coin guarantees that the p-th coin flip at each replica returns an identical value. That is, all the replicas that execute Line 24 in the same phase must obtain the same *state* variable. Such a coin can be easily implemented in our scenario as detailed later in Section 4. Replicas proceed to the next phase if they have not already terminated at Line 22.

The thresholds used ensure that all the non-? values contained in phase-p vote messages are identical. Hence, if replicas execute Line 22 or Line 24, they will see the same v. The beauty of Ben-Or's design, which we borrowed, is that replicas break a tie using a coin flip. Recall that a non-? value v is either 0 or 1. Hence, when replicas flip a common coin at Line 26, it has probability $1/2$ to flip to v and will terminate in the next phase. Section 5 extends this observation to obtain the probabilistic termination property.

Examples: The fast path is three message delays, as illustrated in Figure 2. Section 3.2 mentioned two cases that Weak-MVC is guaranteed to use the fast path. We now present the details. Consider the case when each replica has the same proposal, then Weak-MVC terminates in three message delays, because (i) all replicas have $state = 1$ after the exchange stage; (ii) all replicas have $vote = 1$ after round 1 of phase 1; and (iii) all replicas will execute Line 22 because all the vote messages contain 1. Another case of the fast path is when each replica has a unique proposal. In this case, (i) all replicas have $state = 0$ after the exchange stage; (ii) all replicas have $vote = 0$ after round 1 of phase 1; and (iii) all replicas will execute Line 22 because all the vote messages contain 0.

Weak-MVC might be slow if in every communication step, only roughly half of replicas take a particular branch of the if statements, particularly when half of the replicas execute Line 24 and the other half execute Line 26. A slow-case example with $n = 3$ is as follows: (i) two replicas have $state = 1$ and one has $state = 0$ at the beginning of round 1 in the second stage; (ii) two replicas have $vote = ?$ and one has $vote = 1$ at the beginning of the round 2, because they observed different sets of State messages; and (iii) two replicas execute Line 24 with $state = 1$, and one replica executes Line 26 by flipping a coin and obtains 0 for its *state*. Note that step (iii) is identical to step (i). In other words, no progress has been made. This sequence of events might repeat for a few phases if the coin flip is different from v at Line 26. Section 5 proves that the average latency of Rabia is five message

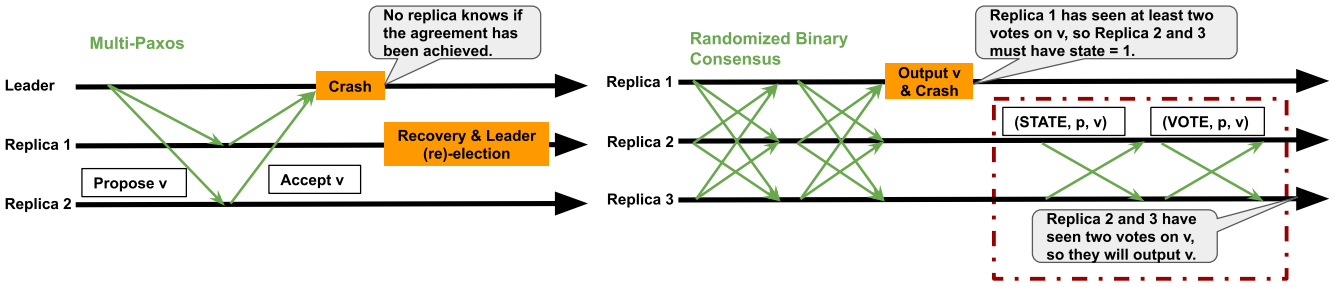


Figure 3. Fail-over in Multi-Paxos vs. No fail-over in Rabia. If the leader in Multi-Paxos fails, then the remaining replicas need to execute a fail-over protocol to recover decisions made by the crashed leader and re-elect a new leader. Rabia does not need any fail-over. If Replica 1 outputs a value v, then the remaining replicas are guaranteed to output v in the next phase, no matter if Replica 1 fails or not. For brevity, only the randomized binary consensus stage of Weak-MVC is shown.

delays. Our evaluation in Section 6 shows that such a long tail latency rarely occurs in a stable network.

3.4 No Fail-Over and Simple Log Compaction

Prior systems, e.g., [25, 38, 39, 43, 48, 52], require a complicated protocol to handle failures, especially when a leader, or a command leader in a multi-leader design, fails. Intuitively, this is because when a leader crashes, replicas need to execute an instance of leader election to ensure liveness. Meanwhile, for safety, replicas need to recover the decision(s) that the previous leader has already made [20, 25, 33]. This task is non-trivial since replicas may have a stale log and may fail again during leader election. In practical systems that optimize throughput with pipelining and leases (for leader or read operations), fail-over becomes even more complicated, because some of the decisions may not have been persisted to a quorum of replicas yet. An example scenario of a crashed leader in Multi-Paxos is depicted in Figure 3.

Rabia does not need any fail-over protocol because it does not rely on a notion of leader, nor command leader. To agree on a value v for a slot, a group of replicas must cast votes for v in some phase of the randomized binary consensus stage; hence, even if some replica(s) fails, a replica can still learn v from the non-faulty replicas in the group. In other words, the use of the VOTE messages ensures that no single replica makes the decision alone, unlike prior algorithms. In Rabia, a group of replicas makes a "joint decision." This design simplifies the failure handling.

The execution on the right part of Figure 3 illustrates the case when there are three replicas, and only Replica 1 observes two VOTE messages of v and executes Line 22 of Algorithm 2. Then, it fails shortly after it learns the output. This scenario is analogous to the case when the only replica that knew the output fails in Multi-Paxos. Suppose Replica 2 and 3 did not learn the decision in Phase 1 and proceed to the next phase as described in Algorithm 2 with $state = v$. This is because they must have observed at least one VOTE message of v in Phase 1. By construction, after the exchange of phase-2 STATE messages, both Replica 2 and 3 update *vote* to v; hence, they will output v in Phase 2. Note that these steps are already specified in Algorithm 2, and Weak-MVC does *not* need an auxiliary protocol to handle failures.

Log compaction is difficult in prior systems because a leader needs to send a snapshot to replicas that lag behind [3, 20]. The issue becomes even more challenging during the process of leader re-election and recovery. In fact, Chubby developers [20] mentioned that "*(Snapshotting) mechanism appears straightforward at first... However, it introduces a fair amount of complexity into the system.*" Particularly, the metadata needs to be stored along with the snapshot itself, which both need to be recovered consistently if a leader failure occurs. Rabia does not need a fail-over and does not rely on the notion of leader; hence, it supports a simple mechanism for log compaction. In particular, each replica has the same responsibility in Rabia, so a slow replica can learn from any other replica to catch up with missing slots in a stable network where no message is lost between non-faulty replicas. Once a slow replica gets to the current slot, it can participate in the agreement process again. If messages could be lost, then log compaction needs to be made more conservative than the one presented in Algorithm 1 – replicas need to make sure that a quorum has already agreed on a slot before compacting a slot.

3.5 Performance Analysis and Trade-off

Rabia does not need a fail-over protocol, but it demands more communication, which we believe is a good trade-off in the target scenario. As we will see in Section 6, Rabia's performance with three replicas in the same availability zone is better than prior systems. Although we observe a degrading performance with a larger n or deployment across different availability zones (in a datacenter), Rabia still has performance comparable with EPaxos.

Fast Path Case: For $n = 3$, we analyze the best-case performance. Both Multi-Paxos and EPaxos (with zero conflict rate) need two message delays, whereas Rabia has three message delays. Mutli-Paxos sends four messages, and EPaxos sends four messages per command-leader. Rabia sends 18

messages. Note that Weak-MVC only sends the original proposal (a request) in the exchange stage and sends a message containing 0, 1 or ? in later phases. That is, bit complexity is dominated by client request size. Therefore, Rabia's communication overhead is not a bottleneck with small n in a stable network.

Average and Worst Case: We prove later in Section 5 that the average latency of Rabia is five message delays on average. As stated earlier, it is possible, in theory, to have a large tail latency due to the slow case of Weak-MVC; however, we argue that such a corner case hardly occurs in practice, as witnessed in our evaluation. In Section 4, we present two practical approaches to mitigate such an unlucky scenario.

Bottleneck: In our evaluation, we found that the bottleneck of Multi-Paxos is the leader. It requires a stable leader to maintain high-performance; however, leader-client and leader-replica communications, especially serialization/deserialization, become the bottleneck. We see 1.5x to 3x difference in throughput between open-loop tests and close-loop tests. As for EPaxos, the bottleneck is local computation time, i.e., dependency check for identifying a permissible order across client requests. Even in the case when there is no conflict, EPaxos still needs to check all the dependencies to ensure safety. The check is proportional to the number of clients, replicas, and the number of client requests in a batch. With 100 clients and batch size 1, the median time for EPaxos to complete the dependency check is 0.29ms, roughly one RTT in the current datacenter. The time increases to 1.90s when batch size increases to 80.

There are two characteristics of Rabia that can potentially reduce performance: message complexity and stable network. We found the trade-off reasonable. Rabia does not need an auxiliary fail-over protocol; moreover, it distributes workload evenly to all replicas, and does not have expensive local computation. As we will see in Section 6, these are why Rabia outperforms Paxos and EPaxos when $n = 3$ in the same availability zone.

4 Practical Considerations

We implemented Rabia in Go, version 1.15.[3] The entire framework, including the key-value storage on top of our system, consists of around 2,200 lines of Go code. Go is a compiled language and has garbage collection and built-in support for managing high concurrency. The implementation follows closely with the pseudo-code. We use go-routines and go-channels for handling connection and communication. This simplifies the logic and increases parallelism.

Common Coin: Rabia uses a common coin (Line 26 in Algorithm 2). We implement it by using a random binary number generator with the same seed across all replicas. This satisfies our purpose by (i) picking 0 or 1 randomly in each phase; and (ii) giving the same value to all replicas in the same phase, i.e., all replicas have the same p-th coin flip. A seed is pre-configured in such a way that for each slot, all replicas use a common seed. When the system is reconfigured, the seeds are reset using a deterministic rule, i.e., the slot index plus the configuration index (epoch number) decide the seed.

The implementation of our common coin is based on the seminal work by Rabin [57] and subsequent works by Michel Raynal [29, 30] on binary randomized Byzantine consensus. Ben-Or's local coin approach [13] tolerates a dynamic adversary, which may determine the set of faulty replicas and delay messages based on the coin flip. The use of a common coin tolerates a weaker adversary which assumes that the faults and message delays are independent of the outcome of the coin flip, which is adequate in practical settings.

Dictionary: At Line 1 of Algorithm 1, Rabia uses a dictionary to keep track of whether the head of the PQ is already in the log or not. Rabia only needs to store a non-$\perp$ $output$ (the output of Weak-MVC) in the dictionary at Line 6, when this $output$ does not match $proposal_i$. TCP implies that the communication is reliable, and a message is delivered exactly once, if the sender is correct. Therefore, replica i is guaranteed to extract $output$ again from its PQ_i when there is no failure. At this step, $output$ can be removed from the dictionary. By assumption, up to f replicas may fail, so PQ_i still has a bounded size if some replicas crash. In our experience, the size of the dictionary is mostly empty in a stable network.

Batching: Following prior work, e.g., [10, 43, 48, 56], we use batching to improve throughput and network utilization. That is, instead of agreeing on one client request for a slot, replicas can agree on a batch of requests. We implement two forms of batching:

- *Proxy batching*: replicas batch a fixed number of client requests before pushing it into the priority queue PQ (Line 8 of Algorithm 1), and forwarding to other replicas (Line 9). Consequently, an entire batch is treated as a proposal at Line 2. If there is no failure, then each batch contains different requests.
- *Client batching*: instead of sending a single request, clients send a batch of requests (i.e., a set of operations) in one message. In practical systems, user applications typically communicate with a load balancing service, which can perform this type of batching by collecting requests from multiple clients [28, 36]. Moreover, as outlined in the experience paper of scaling Memcache by Facebook [51], a query touches 24 keys on average. Such a query can be viewed as a batch of 24 requests to the key-value storage system.

Failure Recovery by Clients: In our design, a client communicates with a single replica, i.e., a proxy replica. If the proxy replica fails, then the client relies on a timeout to detect the unresponsiveness. In such a case, the client re-sends

[3] https://github.com/haochenpan/rabia/

its request to another (randomly selected) replica. To handle the potential duplicated requests in the log, we use the standard solution of embedding a unique ID (a pair of client ID and a sequence number) in each request. When applying requests from the log, a replica skips any duplicated request.

Reconfiguring the Replicas: Since Rabia does not need a fail-over, and all replicas essentially have equal responsibility, it is simple to perform reconfiguration. We treat `add-replica` and `remove-replica` as special commands. A system administrator (or an auxiliary automated membership management component) can submit a special command c to any of the replicas. Replicas will then use Weak-MVC to agree on the slot for c. Eventually, all replicas will learn c, and in the next slot, if c is `add-replica`, then the new replica will join the protocol; otherwise, the removed replica will leave the system. The reason that the reconfiguration is simple compared to prior systems [40, 48, 52] is that Rabia does not rely on the notion of the leader; therefore, every replica is eventually obtaining the same information, which allows them to change to a new configuration jointly. Prior systems need to carefully integrate reconfiguration with leader election to deal with potential leader failures [20, 33].

Tail Latency Reduction: We briefly describe two practical approaches to further reduce tail latency: (i) Using an eventually correct failure detector [35] to allow replicas to receive a consistent set of messages; and (ii) using a freeze time before participating in Weak-MVC, which allows the oldest pending request to be delivered at replicas. The rationale behind these two approaches is to increase the probability of having the same head of each PQ_i for replica N_i, which allows Rabia to take the fast path. However, these two are optimizations that are *not* currently implemented because the network used in our experiments, Google Cloud Platform, shows stability, and hence none of these optimizations should have had any impact on our performance study.

In addition, we adopt a common optimization that allows a slow replica to catch up by asking other replicas when it misses the proposal from a prior slot. That is, a slow replica that learns the decision of a slot may send a request message to other replicas to learn the proposal that has already been agreed upon. This allows the slow replica to participate in the next slot without waiting for delayed messages.

Pipelining: Pipelining is a common optimization that allows the system to proceed to the next slot(s) without learning the output of the current slot. In other words, multiple instances of consensus algorithms are being executed simultaneously. This optimization increases throughput because communication latency is amortized through concurrent slots.

While Rabia's implementation does not include pipelining, our framework can be extended to support it. As presented in Algorithm 1, each replica has one PQ for providing inputs to Weak-MVC. To enable pipelining, we can have multiple PQs for each replica. Then Rabia has one PQ to handle the request batches from a fixed set of replicas, and multiple instances of Weak-MVC can run concurrently and independently. Since randomization ensures that each instance is guaranteed to terminate, liveness still holds with this pipelining strategy.

5 Safety and Liveness Properties

The design of Rabia lends itself to formal proofs of correctness. We briefly discuss how we use Ivy and Coq to formally verify its safety. The complete proof is presented in [54]. We then present a simple analysis of liveness.

Formal Proof of Safety: The safety of Weak-MVC implies the safety of Rabia. Weak-MVC's key safety properties are *weak validity* and *agreement*. In the discussion below, we say that a replica "decides" on a non-? value v in phase p if the replica executes Line 22 of Algorithm 2 after seeing a majority of phase-p VOTE messages with value v. By construction, replicas will decide on only 0 or 1.

Weak validity says that if a replica decides on a value v other than $\perp$, some client must have initially proposed v. Agreement says that if a replica decides on v_1 and another replica decides on v_2, then $v_1 = v_2$. These are the usual properties expected of a correct consensus algorithm, except that weak-validity allows for replicas to decide on $\perp$.

We prove these two safety properties using a combination of the Ivy [46] and Coq [62] verification tools. Ivy is an automated tool that checks whether a property is an *inductive invariant* of a system, meaning that it holds before and after every action of a replica. Automated proof search is achieved by careful restrictions on the types of properties that can be expressed and checked in order to ensure that they fall within a decidable fragment of first-order logic. Meanwhile, Coq is a general-purpose theorem prover, which is highly expressive but requires a human to construct the proof.

The core part of our safety proof is showing that the randomized binary consensus stage of Weak-MVC (Algorithm 2) satisfies agreement. For this, we follow the structure of prior paper proofs of correctness for Ben-Or's algorithm [7]. We first use Ivy to verify that the following four properties are inductive invariants of the system:

1. Any two decisions within a phase must be on the same value v.
2. Once a replica decides on a value v, the next phase is *value-locked* on v, meaning that all the replicas that have neither crashed nor decided must enter the next phase with $state = v$.
3. If a phase is value-locked on v, any decisions within that phase must be for v.
4. If phase i is value-locked on v, then $i + 1$ is also value-locked on v.

We next use Coq to prove that agreement follows from the above four properties by induction on the phase number.

Splitting the proof across the two tools in this way makes use of the relative strengths of each. Checking the four properties above is straightforward in Ivy but would require many lines of proof in Coq. Conversely, trying to do the induction on the phase number within Ivy seems difficult to do while remaining in the supported decidable fragment.[4]

The Ivy part of the proof is 366 lines of code, about 150 of which are a description of Weak-MVC in Ivy's modeling language. The remainder are statements of other intermediate inductive invariants that Ivy uses to establish the four properties above. The Coq file is 190 lines of code, with 128 lines describing the system, axiomatizing the properties checked by Ivy, and stating the agreement and weak validity.

Proof of Liveness: For liveness, we need to show that Weak-MVC terminates with probability 1, as long as the majority of replicas are non-faulty. The proof structure follows from prior works [11, 13, 49, 58]. We first prove the following lemma and use it to prove three key theorems.

Lemma 1. *Each phase has probability at least $\frac{1}{2}$ of leading to termination. That is, all the non-faulty replicas output a value before or at the end of phase p.*

Theorem 1. *Weak-MVC has average round complexity = 5.*

Proof. Lemma 1 implies that the probability of Weak-MVC's Randomized Binary Consensus Stage terminating at the end of phase t is at least

$$(1 - \frac{1}{2})^{t-1}\frac{1}{2} \tag{1}$$

It follows that the number of phases until termination is upper bounded by a geometric random variable whose expected value is $1/2$. Therefore, the average number of phases of this stage is 2. Since each phase has two rounds and Weak-MVC also needs one round in the Exchange Stage, the average number of rounds is $2 \cdot 2 + 1 = 5$. □

Equation (1) implies that Weak-MVC terminates with probability 1. Since Rabia only performs communication in Weak-MVC, the average number of rounds for Rabia to complete a slot is 5, and each slot is completed with probability 1.

6 Performance Evaluation

We first evaluate Rabia by comparing it against two other systems: Multi-Paxos and EPaxos (with no conflicting requests) to understand the performance of the core consensus component. Then, we integrate Rabia with Redis (RedisRabia) and compare it with Redis. Multi-Paxos is a popular choice in production systems [16, 19, 20]. EPaxos (with no conflict) is a state-of-the-art SMR system that achieves the best performance in our setting. It obtains high performance by achieving fast-path latency of two message delays, and each replica can have requests in a different order.

[4]Recent versions of Ivy have some support for including interactive proofs, which might allow doing this proof entirely within Ivy.

The purpose of our evaluation is to understand the performance of the core algorithm with minimum optimization; hence, we choose Multi-Paxos and EPaxos that only use pipelining and batching. Highly optimized systems, e.g., Compartmentalized Paxos [65], and systems that use specialized hardware like NOPaxos [41] could potentially achieve higher performance.

For EPaxos and Multi-Paxos, we use the implementation from [48]. We implement a replicated key-value store that supports read (Get) and write (Put) operations based on Rabia. The structure follows closely to the one from [48] so that we can perform a fair comparison. EPaxos uses its own GoBin library for serialization, which outperforms GoGo Protobuf,[5] the library we used. However, we still choose GoGo Protobuf because it has native support to strings, which allows us to use Redis commands as inputs to Rabia directly. In the EPaxos evaluations, all requests are non-conflicting so that the achieved throughput is the maximum.

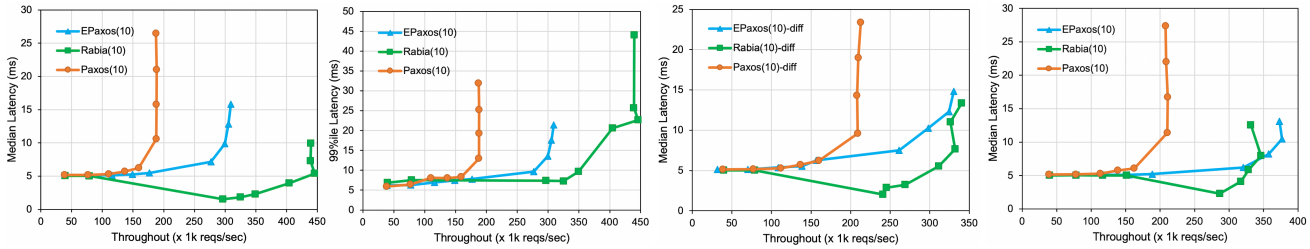
We evaluate our system on the Google Cloud Platform (GCP). Each server is on an e2-highmem-4 instance (Intel Xeon 2.8GHz, 4 vCPU, 32GB RAM) running Ubuntu-1604-xenial-v20201014. Clients run on one to three customized e2 machines, and each machine has 30 vCPUs and 120 GB RAM. All machines are deployed within a single availability zone us-east1-b (except for Figure 4c). Network bandwidth was measured in excess of 7.8 Gbits per second. The typical RTT is about 0.25 ms. The size of a client request is 16B (except for one experiment). We use these sizes because, for example, in Facebook's production TAO system [15, 42], 50% of requests have value field smaller than 16B. In [12], Facebook documents the workload of the Memcached deployment; in one of the systems, 40% of requests is less than 11B. In our setup, we observe no difference between different write ratios. For the data reported, the write ratio is 50% for all three systems.

Performance without Batching: We first consider the case without any form of batching, i.e., both client batch and proxy batch are set to 1, and hence each slot contains one request. We test EPaxos and Paxos, with and without pipelining (labeled with (NP)). As described above, Rabia does not currently implement pipelining, and thus it processes one slot at a time. This closed-loop test consists of three replica machines and one client machine. We vary the load (number of clients) on each system until a maximum throughput is reached. The best performing number of clients varies from two to six clients.[6]

Rabia is comparable with EPaxos (NP). This is because EPaxos spends more time in local computation, whereas Rabia has a higher communication overhead. EPaxos (NP) has a large median latency, compared to pipelined EPaxos,

[5]https://github.com/gogo/protobuf

[6]In this particular setting, EPaxos is bottlenecked by dependency checking. The median latency of the check is 0.29 ms on the CPUs in GCP, which limits EPaxos's throughput. Hence, Paxos outperforms EPaxos in this evaluation.



(a) Median Latency. Same Zone with 3 Replicas.
(b) 99th Percentile Latency. Same Zone with 3 Replicas.
(c) Median latency. Multi-Zones with 3 Replicas.
(d) Median latency. Same Zone with 5 Replicas.

Figure 4. Throughput vs. Latency plots. The deployment consists of 3 or 5 4-CPU replica machines and 3 separate client machines. Both systems have client batch size = 10, which is reported in brackets in the legend.

	Rabia	EPaxos(NP)	EPaxos	Paxos(NP)	Paxos
Thpt	2458.56	2561.3	11480.1	1209.26	12993.07
M-Lat.	1.35	3.99	0.46	2.74	0.67

Table 1. Performance without Batching. (NP) indicates that a system has no pipelining. Throughput is represented as req/s, and median latency is measured in ms.

because it receives more messages before processing each slot, which increases the time for doing dependency check (that is proportional to the requests a command leader has seen). Naturally, the pipelined version outperforms Rabia by 5x. However, Rabia performs well without the pipelining optimization, considering the RTT is roughly 0.25ms. It is close to the maximum number of slots based on the theoretical analysis – each slot takes 1.5 RTT on a fast path, which results into around 2667 slots per second. When batching is used, the effect of pipelining becomes less obvious, as serialization of client requests becomes the bottleneck.

Throughput vs. Latency: We next configure Rabia, EPaxos (with pipelining), and Paxos (with pipelining) to achieve its maximum throughput without reaching saturation and maintain a stable performance. Figure 4 presents the throughput and latency numbers by increasing the number of concurrent closed-loop clients (20 – 500). Interestingly, we found that an optimal configuration is different for each system. EPaxos and Multi-Paxos require proxy batching of size 1000 and 5000, respectively. Rabia runs best with small client batching and proxy batching. In this particular set of tests, Rabia uses proxy batching at 20. All systems use client batching size 10. The maximum batch size is 1000, 5000, and 300, for EPaxos, Paxos, and Rabia, respectively. These numbers are also used in [48]. Following the best practice in [48], each system uses a timeout of 5ms to batch requests if the desired batch size is not reached.

Intuitively, different systems require different parameters to achieve the maximum throughput because each system has a different performance bottleneck. The reasons behind the choices of timeout and batch sizes, based on our experience, are: (i) Paxos has a bottleneck at the leader, so it cannot initiate a slot too frequently; and (ii) Rabia does not implement pipelining, but distributes workload evenly at each replica; hence, it has to agree on one slot more quickly to obtain the maximum throughput. EPaxos falls in the middle.

Figure 4a and Figure 4b show that Rabia has a small increase from the median latency to the 99th percentile latency when all three replicas are deployed in the same availability zone. However, if the system is overloaded, the 99th percentile latency becomes larger because of the increasing chances of NULL slots. Compared to EPaxos and Multi-Paxos, Rabia has a smaller drop in median latency because it uses a smaller proxy batch, and timeout is not triggered when there is a large enough number of clients.

Figure 4c shows that Rabia has a moderate degraded performance, around a 23% drop, when deployed in multiple availability zones in the same region. The deployment has three replicas: one replica in us-east-1-b, one in us-east-1-c, and the other in us-east-1-d. All the clients are deployed in us-east-1-b. The average RTT across multiple zones increases from 0.25 to 0.4ms, and the variance becomes larger at 0.17ms. The limited drop in performance moving from one to multiple availability zones also shows that the network conditions that allow Rabia to reach consensus fast are practical, not stringent, and can be met even when Rabia is deployed beyond a single highly-connected network. Interestingly, EPaxos and Paxos have improved performance. This is because longer RTT actually reduces the communication burden on the leader node (in Paxos) and command-leader (in EPaxos).

Figure 4d presents the case with five replicas. Due to its $O(n^2)$ message complexity, Rabia has reduced throughput. The median latency remains small, compared to other systems. EPaxos improves its performance, as also observed in [48], because when there are no conflicting requests, more requests can be handled concurrently with a larger number of replicas. Figures 4c and 4d demonstrate that Rabia has high performance in the ideal case (same availability zone with $n = 3$), but in other cases, the performance is still comparable to EPaxos.

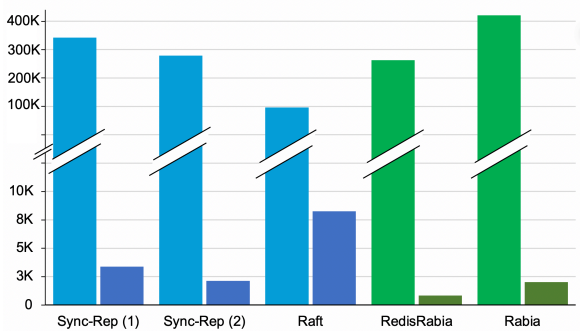


Figure 5. Throughput across different Redis integration. RedisRabia has around 1K req/s without batching, and Rabia has around 2K req/s without batching.

Varying Data Size: Using the same configuration in Figure 4a, we also compare all three systems with key-value pairs with size 256B. EPaxos and Paxos suffer 71% and 56% reduction in throughput, whereas Rabia has less (47%) reduction. Even though Rabia has a higher message complexity for each slot, EPaxos and Paxos rely on pipelining to obtain improved performance, which produces more messages in the same interval and therefore increases network utilization, reaching saturation faster.

Integration with Redis: In our integration, we use Redis to store the key-value pairs instead of a map stored in memory. RedisRabia utilizes Redis native MGet and MPut commands to process a batch of requests. Figure 5 presents the comparison with native Redis-based systems: (i) synchronous-replication with one master and one replica (Sync-Rep (1)); (ii) synchronous-replication with one master and two replicas (Sync-Rep (2)); and (iii) RedisRaft, an experimental system by Redis Labs (Raft). We implement synchronous replication using Redis's WAIT command on top of a native asynchronous replication in Redis (cluster mode). Note that data might be lost or become stale if the master fails in this type of replication. Rabia denotes a system without using Redis as the storage. In all the systems, we turned off any of the persistence options.

Each system has two numbers, throughput with batching (left bar) and without batching (right bar). In all systems, we use 20 for client batching and 15 for proxy batching. The storage engine affects the performance of Rabia significantly because pipelining is not currently implemented, and any delay in completing a slot decreases throughput. Yet, RedisRabia is comparable with synchronous replication. The current implementation of RedisRaft is not optimizing throughput; hence, its throughput is sub-par except for the no-batching case due to its pipelining optimization. On the other hand, RedisRaft has better latency than all other systems, including Rabia-based ones. The reason for it is mainly the effect of pipelining, as also observed in Table 1.

Internal Statistics and Network Stability Test: The performance of Rabia depends on the underlying network, which affects two important internal statistics: (i) the message delays needed for Weak-MVC to terminate; and (ii) the percentage of NULL slots in the log. In summary, the maximum number of message delays we have observed is 15, which occurred 26 times out of 0.12 billion rounds, including the experiments that overload Rabia. 96% of slots use the fast path. For all the closed-loop tests, 2.22% of slots are NULL, whereas for all the open-loop tests, 0.31% of slots are NULL. We also conduct a simple network test in GCP and CloudLab [24] to measure how many messages a replica needs to receive for all three replicas to have the same oldest message in an interval. This scenario ensures that all replicas have the same head in their respective priority queue (PQ). On average, replicas need to receive 3.1 to 3.9 messages. These results, along with the reported internal statistics, confirm that the network conditions of GCP have the stability needed to enable high performance in Rabia.

7 Related Work

SMR: State-machine replication is a popular mechanism for making a service fault-tolerant and highly available [60]. Classical log-based SMR systems use a single leader to order requests, e.g., [6, 37–39, 52]. The capacity of a single node limits these systems. Recent SMR systems adopt a multi-leader (or so-called leaderless) design that distributes the ordering responsibility evenly to increase performance (e.g., Mencius [43], EPaxos [48], M^{2}Paxos [56], Caesar [10], and Atlas [25]). At the core of the multi-leader design [59], there is a dependency graph to keep track of the dependency between requests so that any replica can decide the order of requests fast. As opposed to Rabia, both leader-based and multi-leader-based SMR systems require non-trivial effort [16, 20, 33] to integrate the core consensus algorithms with an auxiliary fail-over mechanism to recover from the failure of a leader or a command leader. Moreover, in the case of multi-leader-based SMRs, replicas also need to spend more computation in checking dependencies. These systems did not use randomized consensus.

Randomized Consensus Algorithms: Following Ben-Or's work, several theoretical papers [5, 11, 26, 29, 30, 49, 57, 58, 67] have improved it in multiple aspects, including tolerating Byzantine faults [5, 49], reducing the average number of message delays using a common coin [5, 26, 57, 58], and taking multi-valued inputs [67]. Ben-Or's algorithm has been previously formally verified in several systems [14, 45]. Pedone et al. [55] investigated two relaxed randomized consensus

algorithms based on Ben-Or's algorithm [13] and Rabin's algorithm [57], and also exploited the weakly ordering guarantees from the network layer to improve performance. Crain [23] presented an evaluation of his binary Byzantine consensus algorithm. The implementation only takes binary input and generates a binary output. These works focused on the single-shot consensus, and did not present an approach to develop a practical SMR system, as Rabia does.

Practical Randomized Byzantine Fault-tolerance Systems: Randomization has been recently used in Byzantine Fault-tolerance (BFT) systems. Cachin et al. proposed a coin-flipping protocol based on the Diffie-Hellman problem, which is then used to build a Byzantine consensus protocol with constant message delays in expectation and has message and communication complexities close to the optimum [17]. Cachin et al. [18] introduced a protocol for cryptographically secure randomness generation that allows users to deploy non-deterministic applications on top of the proposed BFT systems. Miller et al. presented HoneyBadgerBFT [47], which also identified that randomization can be used to improve performance in SMR systems. The key technique is a randomized atomic broadcast protocol that tightly integrates random selection and encryption.

Another popular application of randomized consensus is Blockchain. For example, the Proof-of-Work of Bitcoin [50] and Proof-of-Stake of Algorand [21, 31] can both be viewed as randomized consensus algorithms. These BFT systems focus on cryptographical guarantees [17, 18], permissionless settings [21, 31, 50], or wide-area networks [21, 31, 47, 50]. These aspects are significantly different from our target scenario. Therefore, their techniques, including the ways that randomization is used, are different from Rabia's. Specifically, these systems do not use relaxed consensus (with weak validity) for improving performance.

8 Conclusion

We present the design and implementation of Rabia, a simple SMR system that achieves high performance in favorable settings (e.g., same availability zone with $n = 3$) and comparable performance in more general cases within a single datacenter. By using randomized consensus as opposed to a deterministic one, Rabia does not need any fail-over protocol and supports trivial log compaction. Our evaluation study confirms that in commodity networks, the belief that randomized consensus implementations do not provide competitive performance can be challenged.

Acknowledgment

The authors thank our shepherd Manos Kapritsos and all anonymous reviewers and artifact reviewers for their important comments. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1816487 and CNS-2045976. Early evaluation and artifact evaluation of this work were obtained using the CloudLab testbed [24] supported by the National Science Foundation. The authors would also like to thank Matthew Abbene and Andrew Chapman's help on Redis integration, and Rachel Trickett's help on artifact evaluation.

References

[1] etcd: A distributed, reliable key-value store for the most critical data of a distributed system. https://etcd.io/, accessed May 2021.

[2] RedisRaft: Strongly-consistent Redis deployments. https://github.com/RedisLabs/redisraft, accessed December 2020.

[3] Replication Layer | CockroachDB Docs. https://www.cockroachlabs.com/docs/stable/architecture/replication-layer.html, accessed December 2020.

[4] RethinkDB 2.1: high availability. https://rethinkdb.com/blog/2.1-release/, accessed December 2020.

[5] Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, and Ling Ren. Synchronous byzantine agreement with expected O(1) rounds, expected $o(n^2)$ communication, and optimal resilience. In Ian Goldberg and Tyler Moore, editors, *Financial Cryptography and Data Security - 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers*, volume 11598 of *Lecture Notes in Computer Science*, pages 320–334. Springer, 2019.

[6] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond consensus for microsecond applications. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 599–616, 2020.

[7] Marcos K. Aguilera and Sam Toueg. The correctness proof of ben-or's randomized consensus algorithm. *Distributed Comput.*, 25(5):371–381, 2012.

[8] Kyle Kingsbury (Aphyr). Jepsen: Rethinkdb 2.2.3 reconfiguration. https://aphyr.com/posts/330-jepsen-rethinkdb-2-2-3-reconfiguration. Technical report, 2016.

[9] Kyle Kingsbury (Aphyr). Redis-raft 1b3fbf6. https://jepsen.io/analyses/redis-raft-1b3fbf6. Technical report, 2020.

[10] Balaji Arun, Sebastiano Peluso, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. Speeding up consensus by chasing fast decisions. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017, Denver, CO, USA, June 26-29, 2017*, pages 49–60. IEEE Computer Society, 2017.

[11] James Aspnes. Randomized protocols for asynchronous consensus. *Distrib. Comput.*, 16(2–3):165–175, September 2003.

[12] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In Peter G. Harrison, Martin F. Arlitt, and Giuliano Casale, editors, *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '12, London, United Kingdom, June 11-15, 2012*, pages 53–64. ACM, 2012.

[13] Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, PODC '83, page 27–30, New York, NY, USA, 1983. Association for Computing Machinery.

[14] Nathalie Bertrand, Igor Konnov, Marijana Lazic, and Josef Widder. Verification of randomized consensus algorithms under round-rigid adversaries. In *30th International Conference on Concurrency Theory, CONCUR 2019, August 27-30, 2019, Amsterdam, the Netherlands*, pages 33:1–33:15, 2019.

[15] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C. Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun

Song, and Venkateshwaran Venkataramani. TAO: facebook's distributed data store for the social graph. In Andrew Birrell and Emin Gün Sirer, editors, *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*, pages 49–60. USENIX Association, 2013.

[16] Michael Burrows. The chubby lock service for loosely-coupled distributed systems. In Brian N. Bershad and Jeffrey C. Mogul, editors, *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 335–350. USENIX Association, 2006.

[17] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *J. Cryptol.*, 18(3):219–246, 2005.

[18] Christian Cachin, Simon Schubert, and Marko Vukolic. Non-determinism in byzantine fault-tolerant replication. In Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone, editors, *20th International Conference on Principles of Distributed Systems, OPODIS 2016, December 13-16, 2016, Madrid, Spain*, volume 70 of *LIPIcs*, pages 24:1–24:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[19] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157, New York, NY, USA, 2011. Association for Computing Machinery.

[20] Tushar Deepak Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In Indranil Gupta and Roger Wattenhofer, editors, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC 2007, Portland, Oregon, USA, August 12-15, 2007*, pages 398–407. ACM, 2007.

[21] Jing Chen and Silvio Micali. Algorand: A secure and efficient distributed ledger. *Theor. Comput. Sci.*, 777:155–183, 2019.

[22] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In Chandu Thekkath and Amin Vahdat, editors, *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 251–264. USENIX Association, 2012.

[23] Tyler Crain. A simple and efficient asynchronous randomized binary byzantine consensus algorithm, 2020.

[24] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[25] Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. State-machine replication for planet-scale systems. In Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer, editors, *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 24:1–24:15. ACM, 2020.

[26] Paul D. Ezhilchelvan, Achour Mostéfaoui, and Michel Raynal. Randomized multivalued consensus. In *4th International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2001), 2-4 May 2001, Magdeburg, Germany*, pages 195–200. IEEE Computer Society, 2001.

[27] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[28] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[29] Roy Friedman, Achour Mostéfaoui, and Michel Raynal. Simple and efficient oracle-based consensus protocols for asynchronous byzantine systems. In *23rd International Symposium on Reliable Distributed Systems (SRDS 2004), 18-20 October 2004, Florianpolis, Brazil*, pages 228–237. IEEE Computer Society, 2004.

[30] Roy Friedman, Achour Mostéfaoui, and Michel Raynal. Simple and efficient oracle-based consensus protocols for asynchronous byzantine systems. *IEEE Trans. Dependable Secur. Comput.*, 2(1):46–56, 2005.

[31] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 51–68. ACM, 2017.

[32] Jon Gjengset. Instructors' guide to raft. https://thesquareplanet.com/blog/instructors-guide-to-raft/, 2016.

[33] Tobias Grieger. Consensus, made thrive. https://www.cockroachlabs.com/blog/consensus-made-thrive/, 2016.

[34] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving safety and liveness of practical distributed systems. *Communications of the ACM*, 60(7):83–92, July 2017.

[35] N. Hayashibara, X. Defago, R. Yared, and T. Katayama. The /spl phi/ accrual failure detector. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.*, pages 66–78, 2004.

[36] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 121–136, New York, NY, USA, 2017. Association for Computing Machinery.

[37] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks*, DSN '11, page 245–256, USA, 2011. IEEE Computer Society.

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

[39] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[40] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In Srikanta Tirthapura and Lorenzo Alvisi, editors, *PODC*, pages 312–313. ACM, 2009.

[41] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just say NO to paxos overhead: Replacing consensus with network ordering. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 467–483. USENIX Association, 2016.

[42] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger semantics for low-latency geo-replicated storage. In Nick Feamster and Jeffrey C. Mogul, editors, *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 313–328. USENIX Association, 2013.

[43] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machine for wans. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 369–384. USENIX Association, 2008.

[44] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machine for wans. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 369–384. USENIX Association, 2008.

[45] Ognjen Maric, Christoph Sprenger, and David A. Basin. Cutoff bounds for consensus algorithms. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, pages 217–237, 2017.

[46] Kenneth L. McMillan and Oded Padon. Ivy: A multi-modal verification tool for distributed algorithms. In *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*, pages 190–202, 2020.

[47] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of BFT protocols. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 31–42. ACM, 2016.

[48] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 358–372. ACM, 2013.

[49] Achour Mostéfaoui, Moumen Hamouma, and Michel Raynal. Signature-free asynchronous byzantine consensus with t 2<n/3 and o(n^2) messages. In Magnús M. Halldórsson and Shlomi Dolev, editors, *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, pages 2–9. ACM, 2014.

[50] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009.

[51] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In Nick Feamster and Jeffrey C. Mogul, editors, *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 385–398. USENIX Association, 2013.

[52] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, page 305–320, USA, 2014. USENIX Association.

[53] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos made EPR: decidable reasoning about distributed protocols. *Proc. ACM Program. Lang.*, 1(OOPSLA):108:1–108:31, 2017.

[54] Haochen Pan, Jesse Tuglu, Neo Zhou, Tianshu Wang, Yicheng Shen, Xiong Zheng, Joseph Tassarotti, Lewis Tseng, and Roberto Palmieri. Rabia: Simplifying state-machine replication through randomization, 2021 https://arxiv.org/abs/2109.12616.

[55] Fernando Pedone, André Schiper, Péter Urbán, and David Cavin. Solving agreement problems with weak ordering oracles. In Fabrizio Grandoni and Pascale Thévenod-Fosse, editors, *Dependable Computing - EDCC-4, 4th European Dependable Computing Conference, Toulouse, France, October 23-25, 2002, Proceedings*, volume 2485 of *Lecture Notes in Computer Science*, pages 44–61. Springer, 2002.

[56] Sebastiano Peluso, Alexandru Turcu, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. Making fast consensus generally faster. In *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016, Toulouse, France, June 28 - July 1, 2016*, pages 156–167. IEEE Computer Society, 2016.

[57] Michael O. Rabin. Randomized byzantine generals. In *24th Annual Symposium on Foundations of Computer Science, Tucson, Arizona, USA, 7-9 November 1983*, pages 403–409. IEEE Computer Society, 1983.

[58] Michel Raynal. *Fault-Tolerant Message-Passing Distributed Systems - An Algorithmic Approach*. Springer, 2018.

[59] Tuanir França Rezende and Pierre Sutra. Leaderless state-machine replication: Specification, properties, limits. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, volume 179 of *LIPIcs*, pages 24:1–24:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[60] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[61] Marcelo Taube, Giuliano Losa, Kenneth L. McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, and Doug Woos. Modularity for decidability of deductive verification with applications to distributed systems. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 662–677, 2018.

[62] The Coq Development Team. *The Coq Proof Assistant, version 8.12.0*, July 2020.

[63] Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Comput. Surv.*, 47(3), February 2015.

[64] Klaus von Gleissenthall, Rami Gökhan Kici, Alexander Bakst, Deian Stefan, and Ranjit Jhala. Pretend synchrony: synchronous verification of asynchronous distributed programs. *Proc. ACM Program. Lang.*, 3(POPL):59:1–59:30, 2019.

[65] Michael Whittaker, Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, Neil Giridharan, Joseph M. Hellerstein, Heidi Howard, Ion Stoica, and Adriana Szekeres. Scaling replicated state machines with compartmentalization. *Proc. VLDB Endow.*, 14(11):2203–2215, 2021.

[66] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 357–368, Portland, OR, June 2015.

[67] Jialin Zhang and Wei Chen. Bounded cost algorithms for multivalued consensus using binary consensus instances. *Inf. Process. Lett.*, 109(17):1005–1009, 2009.