Racos: Improving Erasure Coding State Machine Replication using Leaderless Consensus

Jonathan Zarnstorff
Unaffiliated
Boston, MA, USA
jonathanzarnstorff@gmail.com

Lucas Lebow
Unaffiliated
Denver, CO, USA
lucasphone75@gmail.com

Christopher Siems
Clark University
Worcester, MA, USA
lucasphone75@gmail.com

Dillon Remuck
Clark University
Worcester, MA, USA
dremuck@clarku.edu

Colin Ruiz
Clark University
Worcester, MA, USA
cruiz@clarku.edu

Lewis Tseng*
UMass Lowell
Lowell, MA, USA
lewistseng@acm.org

Abstract

Cloud storage systems often adopt state machine replication (SMR) to ensure reliability and availability. Most SMR systems use "full-copy" replication across all nodes, which leads to degraded performance for data-intensive workloads, due to high disk and network I/O costs. Erasure coding has recently been integrated with leader-based SMR systems to reduce the costs, e.g., RS-Paxos, CRaft, HRaft, and FRaft. However, these systems still have *bottlenecks at the leader*, limiting their performance when handling large datasets.

To address the bottlenecks, this paper proposes Racos, which integrates erasure coding with a recent *leaderless* SMR protocol, Rabia Unlike Paxos or Raft, Rabia uses a leaderless design for reaching consensus, making it suitable for our purpose. Compared to a leader-based design, Racos distributes workload evenly, alleviating the bottlenecks.

We integrate our system Racos with etcd, a distributed key-value storage that powers many production systems including Kubernetes. Our evaluation, using YCSB, shows that Racos outperforms the closest competitors by **up to 2.26x** in throughput within local-area networks and reduce median latency by up to **76.8%** in wide variety of workloads.

*This material is based upon work partially supported by the National Science Foundation under Grant CNS-2449640.

SoCC '24, November 20–22, 2024, Redmond, WA, USA

ACM ISBN 979-8-4007-1286-9/24/11
https://doi.org/10.1145/3698038.3698511

CCS Concepts

• **Computer systems organization** → **Reliability**.

Keywords

State machine replication, Erasure coding, Leaderless, Rabia

ACM Reference Format:
Jonathan Zarnstorff, Lucas Lebow, Christopher Siems, Dillon Remuck, Colin Ruiz, and Lewis Tseng. 2024. Racos: Improving Erasure Coding State Machine Replication using Leaderless Consensus. In *ACM Symposium on Cloud Computing (SoCC '24), November 20–22, 2024, Redmond, WA, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3698038.3698511

1 Introduction

Modern distributed systems in clouds often use state machine replication (SMR) [43] to provide reliability and availability. SMR allows a collection of nodes (or servers) to behave as a coherent group, even in the presence of asynchrony, node failures, and concurrent operations from the clients. Typically, SMR-based systems support strong consistency, which simplifies and lowers the development complexities of the upper layer applications [7, 12, 14].

Multi-Paxos [29] and Raft [38] are two most popular approaches for implementing the SMR component in production systems, e.g., Google Chubby [6], Google Spanner [14], Microsoft Azure Storage [7], Microsoft Gaios [4], Facebook's LogDevice [32], CockroachDB, and etcd. Both Multi-Paxos and Raft adopt the "*full-copy*" replication, which replicates a complete copy of data to all nodes.[1] As a result, an n-way replication incurs n times of the storage cost in the entire system, compared to a non-replication system.

[1]Some SMR solutions reduce redundancy by replicating to only a subset of nodes, e.g., [16, 19, 31]. However, these approaches are not as common in production systems, due to their complexities. Therefore, we focus on the case when data is replicated to all nodes. Sharding is an orthogonal approach that can be used along with SMR. This paper is focused on SMR; hence, does not consider sharding.

For modern workloads in highly reliable and available systems, full-copy replication solutions are facing performance issues, mainly due to two following factors:

- Most cloud systems need to handle data-intensive workloads with ever-growing data size. The larger the data size, the larger the storage and network costs when using full-copy replication.
- For reliability and durability, many production systems (e.g., etcd and Azure) choose to flush data to disk, before acknowledging a write (or an update) operation. Disk I/O quickly becomes the performance and scalability bottlenecks.

Recent SMR systems, such as RS-Paxos [34] and CRaft [51], adopt erasure coding to reduce costs, and show improved throughput and latency over the full-copy replication solutions. In general, these systems use Reed-Solomon (RS) codes [24], a form of erasure coding, to reduce the amount of data that needs to be stored at the follower nodes to reduce the bottlenecks. In both RS-Paxos and CRaft, followers only need to store a coded segment for a particular data. The size of the coded segment is usually only a fraction of the original data, reducing the I/O costs.

More concretely, writing M units of data into a full-copy system requires M units of storage space and incurs M units of disk I/O on *each* node in a leader-based full-copy SMR such as Multi-Paxos and Raft. In contrast, in RS-Paxos or CRaft, the follower nodes only need to incur M/k units of storage space and disk I/O, where k is a configurable parameter that will be introduced later in Section 2. Lower I/O improves both throughput and latency [34, 51].

However, combining a leader-based SMR and erasure coding brings three main limitations that impact performance, availability, and scalability:

- Only the leader is performing the operations related to erasure coding, limiting performance and scalability.
- In the phase of a slow or saturated leader, the availability is impacted, as only the leader can serve client operations.
- During the phase of leader failover, the new leader needs to perform recovery (which reconstructs each piece of data using coded segments fetched from other followers). The recovery in prior erasure coding SMR takes much longer than the recovery in full-copy systems.[2] In leader-based erasure SMR systems, the reconstruction time is proportional to the size of the entire system. No operation can be performed during the recovery, causing a loss of availability.

[2]In Paxos-based full-copy systems, a new leader only needs to fetch missing data from followers. In Raft-based systems, a new leader does *not* have any missing data, as Raft requires the new leader to have the most up-to-date information (or log entries).

To the best of our knowledge, all the prior erasure coding SMRs [25, 34, 51, 54] are facing these three limitations. This is because in their designs, the leader still needs to store the original data, *not* a coded segment.

Motivated by the observations, this paper aims to answer:

> How do we build an erasure coding SMR that improves performance, availability, and scalability by further alleviating bottlenecks at the leader?

In particular, we adopt a leaderless design so that *each node* only needs to store one coded segment. We empirically demonstrate that our design improves performance, availability, and scalability compared to prior full-copy and erasure coding SMR systems.

Targeted Systems and Workload. In this paper, we focus on distributed storage systems that use SMR within a single datacenter. Following prior works on erasure coding SMRs [25, 34, 51, 54], we focus on the case when each operation is either a *write* or a *read* on a single key. Such an assumption is typical in many key-value storages such as etcd, Redis, and Memcached [36]. Our design can be extended to more general settings (e.g., a transaction operation that touches multiple keys).

In addition, the storage system provides *durability* by flushing data to disks. Many production storage systems provide the option of syncing to disks for increased durability, e.g., Spanner, Azure, Cassandra [28], and CockroachDB. A popular open-source system that always performs disk flushes (before acknowledging a write operation) is etcd, which is a distributed key-value store used by many systems such as Kubernetes and Tencent Cloud Service. We integrate our design with etcd and use YCSB [13] for evaluation.

Traditionally, erasure coding is used for large data for production systems [7, 7, 12, 23, 27, 40]. In this paper, we demonstrate that using our design, erasure coding can also be beneficial for handling small and medium data (at the scale of bytes, kilobytes (KBs) and megabytes (MBs)). Note that etcd is commonly used for storing configuration files. Kubernetes also uses etcd to record events that occur in its cluster. The size of the configuration files and Kubernetes Event objects is typically in the range of KBs, or even MBs. Therefore, our evaluation is focused on data size in this range.

We stress that our design can scale to larger data; however, when dealing with data at the scale of gigabytes (GBs) or more, the bottlenecks are usually not at the SMR layer. Whether a novel SMR design is necessary at this scale is left as interesting future work.

Our System: Racos. To avoid the leader bottlenecks identified earlier, we develop our system based on a leaderless SMR design in which all nodes behave identically. We choose

Rabia [39] as our base SMR protocol; hence, we name our system Racos (RAbia COded System). Each node (or replica) in Racos has storage cost and disk I/O equal to M/k – the same load as the follower nodes in RS-Paxos and CRaft and $1/k$ of their leader node. This is the main reason that Racos has improved performance and scalability over prior leader-based erasure coding SMRs.

Due to the leaderless design, Racos does not need to deal with a saturated leader nor perform leader recovery, which leads to higher availability. Section 2 also discusses why building up top of Rabia, instead of another leaderless SMR, further improves availability.

Racos: Key Techniques. In addition to the careful integration of erasure coding and Rabia, we also introduce two practical optimizations to improve over the original Rabia:

- We modify Rabia's consensus component to reduce the fast-path latency for achieving consensus from 1.5 RTTs (round-trip times) in Rabia to 0.5 RTT in Racos. Note that 0.5 RTT is equivalent to one message delay in the network. Hence, on the fast path, nodes can determine the order of an operation, after one round of exchanging messages with other nodes in the consensus component.[3]
- We increase the chance of staying on the fast path by using a quorum-based approach for reads. Our approach effectively reduces conflicts in the consensus component, compared to Rabia. This results in higher throughput and lower tail latency when there are sufficiently number of writes (which lead to conflicts that degrade Rabia's performance).

Racos: Performance Highlights. We evaluate Racos in local-area networks (LAN), against the optimized version of Raft (the original protocol used in etcd), RS-Paxos [34] and Rabia [39]. We implement Racos, RS-Paxos, and Rabia into etcd. That is, we replace etcd's consensus layer by these three SMR protocols. Our evaluation using YCSB [13] shows that Racos improves performance in both throughput and latency in all the workloads we tested.

In particular, in the all-write highly skewed workload with larger data, Racos has **2.13x-2.21x** throughput improvement and reduces median latency by around **76.8%** over the closest competitors when the systems saturate. In the balanced highly skewed workload, Racos has **1.81x-2.26x** throughput improvement and reduces median latency by **55.8%**.

To our knowledge, Racos is the first leaderless erasure coding SMR design that shows clear performance advantage over existing leader-based designs in a real-world system with various practical workloads.

Outline: We first discuss closely related works on erasure coding SMRs and our motivation in developing a leaderless erasure coding SMR system in Section 2. We then discuss Rabia and its limitations in Section 3. Our design of Racos and its analysis are presented in Section 4. We evaluate Racos in Section 5. More related work is discussed in Section 6.

[3] At first glance, 0.5 RTTs could be surprising. However, the communication pattern in Rabia and Racos is quite different from the leader-based systems like Multi-Paxos and Raft, as both Rabia and Racos separate the spreading of operations and the consensus. Moreover, as in prior works [25, 34, 39, 51, 54], we do not count the communication between clients and nodes here. More details are presented in Section 4.

2 Why Leaderless Erasure Coding SMR?

This section discuss erasure coding SMR and provides the motivations of using a leaderless design.

2.1 Erasure Coding and Reed-Solomon Code

An erasure code [24] is a form of forward error correction code. It was designed under the assumption of bit erasures, i.e., when certain bits might be corrupted or missing (due to disk or storage failures or lost messages). Therefore, erasure code has wide applications in network transfer and distributed storages. An erasure code process transforms a data of k symbols into a larger data with n symbols (i.e., $n > k$) such that the original data can be recovered from *any* subset of the n symbols.

In this paper, we focus on Reed-Solomon (RS) codes [24], which is used by all prior erasure coding SMR systems. An RS code has two parameters k and m, both positive integers. In using a (k, m) code, each piece of data is divided into k "data segments" with equal sizes. These k data segments are then used to generate m "parity segments," through *encoding*. In total, there are $k+m$ segments such that any k out of these $k + m$ segments (namely "*coded segments*") are sufficient for reconstructing the original data, through *decoding*. In our design, each node is assigned to store *exactly* one single data or parity segment; hence, $k + m = n$, which is the number of nodes in the SMR system. As per design of RS codes, the size of one coded segment is $\frac{1}{k}$ of the size of the original data.

2.2 Erasure Coding in Clouds

Erasure coding has been widely adopted in real-world cloud storage and file systems, e.g., Microsoft Azure [7, 23], Microsoft Giza [12], HDFS [20], Google Cloud Storage [37], Facebook's Tectonic Filesystem [40], Oracle's FSS [27], etc. This is mainly because erasure coding allows the systems to explore the tradeoff among storage cost, availability, and reliability. Such an economic argument and operation flexibility are appealing to cloud providers in the era of ever-increasing data volume and velocity. For example, Google Cloud Storage uses erasure coding to achieve at least 99.999999999% annual durability (or 11 nines) without prohibitive costs [37].

Table 1: ***Cost comparison for M unit-cost writes.*** **To tolerate f crashes, a full-copy SMR has $n = 2f + 1$, whereas a erasure coding SMR using (k, m) Reed-Solomon code requires $n \geq 2f + k$.**

	Racos	Full Copy Replication SMR		Leader-based Erasure Coding SMR	
	Node	Leader	Follower	Leader	Follower
Storage Cost	$\frac{M}{k}$	M	M	M	$\frac{M}{k}$
Disk I/O	$\frac{M}{k}$	M	M	M	$\frac{M}{k}$
Network Cost	$\frac{M}{n}(1+\frac{n-1}{k})+\frac{M(n-1)}{n}\frac{1}{k}$	$M \cdot n$	M	$M(1+\frac{n-1}{k})$	$\frac{M}{k}$

These cloud systems are designed to handle larger data or files in the range of several megabytes (MBs), gigabytes (GB), or more (e.g., blob or data warehouse storage). These systems typically implement state machine replication (SMR) and erasure coding separately. In other words, SMR is a standalone component *without* the direct interaction of erasure coding in cloud storages for larger data.

2.3 Erasure Coding SMRs

It was not until recently that erasure coding SMR was introduced to handle smaller data, from kilobytes (KBs) to MBs. The novelty is on the *integration* of erasure coding with an SMR protocol. RS-Paxos [34] integrates erasure coding with Paxos. CRaft [51] integrates coding with Raft. Both systems outperform the original protocols in local-area networks, in common cases when the network is reliable and the leader is responsive. HRaft [25] and FlexRaft [54] improve the performance, over RS-Paxos and CRaft, in non-common cases, by designing a more flexible coding scheme that is adaptive to node failures or message losses. CRaft, HRaft and FlexRaft assume accurate failure detectors to have good tradeoff among storage cost, performance, and resilience. (More discussion in Section 6.) We focus on the common-case performance in this paper, and do not assume the existence of accurate failure detector.

All these erasure coding SMR systems adopt a leader-based design in which the leader node still needs to store the full copy of data; hence, these systems suffer the limitations identified in Section 1. These systems have generally stable and good performance when the leader handles light load. However, both performance and availability are impacted when the leader is saturated. As availability is the first-class citizen in almost all modern cloud storage systems, these erasure coding solutions are not as appealing for production systems. In fact, in most of our evaluations, RS-Paxos performs slightly worse than Raft. This motivates us to explore a *leaderless* design.

2.4 Leaderless Erasure Coding SMR

We choose Rabia [39] as our base SMR protocol; hence, we named our system **Racos (RAbia COded System)**. In Racos, all nodes behave identically. Each node stores and processes the same amount of data as the followers in prior leader-based erasure coding SMRs. This feature effectively reduces the aforementioned bottlenecks.

First, we analytically show that a leaderless solution like Racos provides better performance by lowering the disk and network I/O's. Table 1 presents the cost comparison for different types of SMR systems. Each system consists of n nodes and handles M writes from the clients. For simplicity, assume each write operation carries 1 unit of data. In the (leader-based) full-copy SMR, the leader needs to forward the writes to followers; hence, the network cost is $M \cdot n$, including the communication with the clients.

Consider erasure coding SMRs that use an (k, m) Reed-Solomon codes. Recall that by design, $n = k + m$ and each coded segment has the size of $1/k$ units. (Coded segments include both data and parity segments.) Prior leader-based systems (e.g., [25, 34, 51, 54]) has the leader store the full copy of the data, which leads to storage cost and disk I/O equal to M/k for the follower and M for the leader. For each write, the leader needs to obtain the original data from the client and forward the coded segments to followers, resulting in $M(1 + \frac{n-1}{k})$ network costs.

In Racos, each node has storage cost and disk I/O equal to M/k for M writes. This is because, in our design, the workload of forwarding segments and communicating with clients are distributed evenly. In addition, the network cost for each node is

$$\frac{M}{n}(1+\frac{n-1}{k})+\frac{M(n-1)}{n}\frac{1}{k}=\frac{M}{n}(1+\frac{2(n-1)}{k}).$$

The first part denotes the amount of work in directly communicating with the clients and forwarding the coded segments, as each node directly handles M/n of writes. The second part indicates the amount of coded segments received from other nodes.

Recall that in our targeted systems (mentioned in Section 1), disk I/O is the major bottleneck for larger data with respect to the common-case performance. Therefore, Table 1 indicates that a leaderless solution has a clear advantage in performance over leader-based solutions. In the case when

network is the bottleneck, our leaderless design is also more beneficial (albeit smaller), because of the evenly-distributed communication.

The liveness of leader-based systems depends on the responsiveness of the leader. In the case of a saturated leader (which occurs more frequently for data-intensive workloads), the system availability and scalability are both affected. In contrast, there is no single bottleneck in typical leaderless designs, leading to higher availability and scalability.

In addition, the expensive failover of leader-based erasure coding SMR brings a concern of unavailability. Recall that the newly-elected leader needs to reconstruct all the data in the system, before it can serve new operations. Consider the "Reed-Solomon Erasure Coding in Go" library[4] that we use in our implementation. It is an optimized Go port of the JavaReedSolomon library released by Backblaze [2]. The library has throughput around 1 GB/s. A typical etcd cluster has data size around 2 GBs. This means that during the failover, the system will be unresponsive for 2 seconds in the ideal case (without taking the time for electing a new leader and for gathering coded segments into consideration). The unavailability is worse for larger clusters. A leaderless design avoids this issue.

Why Rabia? Next, we explain why Rabia [39] is an appropriate choice for our targeted workloads. Recently, numerous leaderless SMR designs are proposed in the literature, e.g., [15–17, 30, 33, 39, 42]. We choose Rabia mainly due to the following reasons:

- *Designed for an Individual Datacenter*: Most leaderless solutions are optimized for wide-area networks (WANs), such as EPaxos [33], Atlas [17], and Tempo [16]. They adopt some assumptions that are not suitable for our targeted systems. For example, Atlast and Tempo assume that the number of concurrent failures is very small, which is appropriate for WANs, as failures or unresponsiveness of individual datacenters are relatively rare. In LANs, it is important to tolerate concurrent failures or unresponsive nodes.
- *Availability*: One key subtle observation is that Rabia is based on an asynchronous consensus protocol, which is related to higher availability. First, other leaderless solutions like EPaoxs, Atlas, and Tempo all use the idea of dependency tracking to resolve conflicts for writes. As observed in many prior works [5, 44, 47, 48], dependency tracking can easily lead to either long tail latency (when resolving a long chain of dependencies) or a complicated recovery protocol. In contrast, Rabia does not use dependency tracking.

  Second, due to Rabia's property, we can design reads so that they are "*non-blocking*," using a quorum-based approach. More concretely, in Section 4.3, we show that Racos's read can always complete, as long as the client can communicate with a quorum of nodes. Note that in prior designs, either leader-based or leaderless ones, reads could be unavailable during the leader failover or dependency resolving.

[4]Reed-Solomon Erasure Coding in Go. https://github.com/klauspost/reedsolomon

3 Primer on Rabia and its Limitations

The key functionality of SMR [43] is to agree on the *ordering* of the operations from the clients. Each node maintains a log of operations that follows the agreed order, which implies linearizability (or strong consistency). A key challenge is to reach consensus on the agreed order efficiently.

Rabia: Overview. Rabia [39] is a recent leaderless SMR system designed for a single datacenter. Rabia can achieve agreement under asynchrony without a fail-over protocol nor dependency tracking because of its usage of Ben-Or's randomized binary consensus (RBC) [3]. Rabia uses RBC to agree on ordering of the log.

More concretely, Rabia consists of (i) a spreader that exchanges client operations among nodes; and (ii) a consensus component that takes 1.5 RTTs to complete on the fast path and 4.5 RTTs on average. Rabia is leaderless; thus, each node can serve client operations directly.

In the consensus component, each node proposes an operation for each slot and runs RBC. If RBC returns true, then it is guaranteed that at least a majority of nodes are proposing the same operation for this slot, which will later be included in the particular slot of log. If RBC returns false, then the slot is "forfeited" and the slot will be skipped (during operation execution), by inserting a null operation to the slot.

Recall that the famous FLP impossibility [18] implies that RBC and Rabia cannot have deterministic termination, in the presence of failures and asynchrony. Indeed, they only provide probabilistic termination guarantees. However, due to the design, Rabia can perform well in a single datacenter with a limited number of nodes.

Rabia: Limitations. Prior works [9, 45, 46] point out a few concerns about deploying Rabia under practical workloads, such as no natural support for pipelining (due to the usage of RBC) and poor scalability (due to its all-to-all communication pattern).[5] The main limitation of using Rabia for our targeted workloads is that Rabia's performance drops significantly (up to 50%) from an all-read workload to a read-intensive

[5]Interestingly, as will be shown in our evaluation, when the disk I/O becomes the main bottleneck, Rabia actually has performance slightly better than both Raft and RS-Paxos when integrated with etcd, even in the case without using batching. It has good scalability in our evaluation as well.

Algorithm 1 Racos **without** Quorum Read: Code for Node N_i

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

Code for Node N_i:
/* Consensus Component */
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-MEC
3: $output \leftarrow$ Weak-MEC($proposal_i, seq$)
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: /* Execution Component */
8: **if** $output = \langle \text{WRITE}, ID_w, c, key_w, Code_w[i] \rangle$ **then**
9: Persist $output$ to disk
10: Return acknowledgement to client c if i is c's proxy
11: **else if** $output = \langle \text{READ}, ID_r, c, key_r \rangle$ **then**
12: **if** i is client c's proxy **then**
13: $ID_r \leftarrow$ ID of the most recent write on key_r in log
14: Send read request with ID_r to all nodes
15: **wait until** receiving k coded segments corresponding to ID_r ▷including i's coded segment
16: $data_r \leftarrow$ Decode(received coded segments)
17: Return $data_r$ to client c
18: $seq \leftarrow seq + 1$

/* Spreader Component: executing in background */
Upon receiving $\langle \text{WRITE}, ID_w, c, key_w, data_w \rangle$ **from client** c:
19: $Codes_w \leftarrow$ Encode($data_w$) ▷$Codes_w$ is an n-element array
20: Send $\langle \text{WRITE}, ID_w, c, key_w, Codes_w[j] \rangle$ to node N_j
21: **wait until** receiving $\geq n - f$ acknowledgements
22: Send $\langle \text{WRITEREADY}, ID_w \rangle$ to all nodes
23: $PQ_i.push(\langle \text{WRITE}, ID_w, c, Codes_w[i] \rangle)$

Upon receiving $\langle \text{WRITE}, ID_w, c, key_w, data_w \rangle$ **from node** N_j:
24: Add $\langle \text{WRITE}, ID_w, c, key_w, data_w \rangle$ to buf_i

Upon receiving $\langle \text{WRITEREADY}, ID_w \rangle$ **from node** N_j:
25: $writeOp \leftarrow$ the write op with ID_w in buf_i ▷each write has a unique ID
26: $PQ_i.push(writeOp)$

Upon receiving $\langle \text{READ}, ID_r, c, key_r \rangle$ **from client** c:
27: Send $\langle \text{READ}, ID_r, c, key_r \rangle$ to all nodes
28: $PQ_i.push(\langle \text{READ}, ID_r, c, key_r \rangle)$

Upon receiving $\langle \text{READ}, ID_r, c, key_r \rangle$ **from node** N_j:
29: $PQ_i.push(\langle \text{READ}, ID_r, c, key_r \rangle)$

Algorithm 2 Quorum Read

Code for Node N_i:
Upon receiving $\langle \text{READ}, ID_r, c, key_r \rangle$ **from client** c:
1: Send $\langle \text{GETSLOT}, key_r \rangle$ to all nodes
2: **wait until** receiving $\geq n - f$ responses
3: $slot_r \leftarrow$ the largest received slots
4: Send $\langle \text{GETSEGMENT}, slot_r \rangle$ to all nodes
5: **wait until** receiving k coded segments corresponding to ID_r ▷including i's coded segment
6: $data_r \leftarrow$ Decode(received coded segments)
7: Return $data_r$ to client c

Upon receiving $\langle \text{GETSLOT}, key_r \rangle$ **from node** N_j:
8: $slot \leftarrow$ the slot that contains the most recent write on key_r
9: Return $slot$ to node N_j

Upon receiving $\langle \text{GETSEGMENT}, slot_r \rangle$ **from node** N_j:
10: $CodedSeg_i \leftarrow Codes_w[i]$ stored in $log[slot_r]$
11: Return $CodedSeg_i$ to node N_j

workload with 95% reads. Raft and RS-Paxos have degraded performance, but not as dramatically.

To address this performance issue, we identify two key points for improvement.

C1. How do we mitigate the impact of no-pipelining?
C2. How do we stay on the fast-path as often as possible?

Note that in our targeted scenarios, batching data at KB or MB scales can easily lead to high tail latency due to the computation and communication overhead. Therefore, the batching technique used in [39] is not feasible in our case.

4 Racos: Design and Analysis

We present our design, pseudo-code and analysis in this section.

4.1 Racos: Design Overview

To address challenge C1, we first observe that *not all* messages need to be treated equally in our targeted workloads. More specifically, if we separate the spreader (which sends a corresponding coded segment to each node), then all the other messages (e.g., acknowledgements or messages for the consensus protocol) are small in size, and have small impacts in performance. In Racos, spreader supports pipelining. The only part that has to be performed sequentially is the consensus component which only exchanges small messages.

To address challenge C2, we observe that when using erasure coding, the number of nodes is usually much larger than f. Economical benefits of such a choice are also highlighted in Microsoft Giza (the backend of OneDrive) [12]. Based on this observation, we modify Rabia's consensus component to shorten the fast path.

Another important limiting factor for Rabia under high loads is when each node proposes different operations (i.e., *conflicting operations* are proposed for the same slot). If this

occurs, the RBC will return false, leading to a NULL operation that reduces performance. We use two approaches to mitigate the limitation. First, the spreader is designed in a way that only the operations that are received by a sufficient number of nodes (i.e., $k + f$) will be used as candidates for the consensus component. This reduces the chance of nodes proposing different operations. Second, we introduce Quorum Reads for Racos such that only writes need to be ordered by the consensus component. This reduces the chance of conflicts.

In short, to address C1, we (i) separate spreader and consensus so that spreader can exchange larger messages (that contain coded segments) in parallel; and (ii) use a quorum-based approach to process reads in parallel.

To address C2, we first identify a condition to shorten the fast path. The consensus component fast-path latency is reduced from 1.5 RTTs in Rabia to 0.5 RTT in Racos. Second, we use the quorum-based approach to reduce conflicting proposals. Consequently, the nodes can stay on the fast path more frequently.

4.2 Racos: Pseudo-code

Algorithm 1 presents the pseudo-code of Racos. Algorithm 3 presents the pseudo-code for the key consensus component, which allows us to use RBC (randomized binary consensus) to agree on the ordering of *coded segments* efficiently. Algorithm 4 presents the helper function. Finally, Algorithm 2 presents the pseudo-code for using a quorum-based approaching for reading a particular data.

Racos follows the structure of Rabia; however, as mentioned earlier, we changed a few key components. The new changes are highlighted in the pseudo-code. In particular, we make the following changes to integrate Rabia with erasure coding:

- *Separation of consensus and spreader*: Line 18 to Line 27
- *New consensus component*: Line 3
- *Separation of execution and consensus*: Line 8 to Line 17
- *Quorum Read*: Algorithm 2

Consensus Component. Each node uses a priority queue to keep track of unprocessed operations (including the ones forwarded by other nodes). As assumed in many prior works (e.g., [39]), each operation has a unique identifier ID and a machine timestamp. The priority queue uses the timestamp to order these pending operations. Note that similar to Rabia, Racos's correctness does not rely on clock synchronization. The timestamp is purely within the priority queue. In particular, the consensus component does *not* rely on this timestamp. Thus, the liveness of the system does not rely on the accuracy of clock synchronization. The timestamp is omitted in the pseudo-code for brevity.

At Line 2 of Algorithm 1, each node N_i takes the head of the priority queue $proposal_i$ and uses it as the input of our consensus protocol Weak-MEC, which is specifically designed for erasure coding storages.

The output of Weak-MEC is either a client operation (reads or writes) or a NULL operation:

Algorithm 3 Weak-MEC: Code for Node i

When Weak-MEC is invoked with input q and seq:
1: // Exchange Stage (Phase 0): exchanging proposals
2: Send (Proposal, q) to all
▷q is client c's op of the form $\langle *, ID, c, key, * \rangle$
3: **wait until** receiving $\geq n - f$ Proposal messages
4: **if** ID appears $\geq \lfloor \frac{n}{2} \rfloor + f + 1$ times in received Proposal's **then**
5: **Return** FindReturnValue(1) ▷Fast-path Termination
6: **else if** ID appears $\geq \lfloor \frac{n}{2} \rfloor + 1$ times in received Proposal's **then**
7: $state \leftarrow 1$
8: **else**
9: $state \leftarrow 0$
10: // Randomized Binary Consensus Stage (Phase $p \geq 1$)
11: $p \leftarrow 1$ ▷Start with Phase 1
12: **while** *true* **do**
13: /* Round 1 */
14: Send (State, $p, state$) to all ▷$state$ can be 0 or 1
15: **wait until** receiving $\geq n - f$ phase-p State messages
16: **if** value v appears $\geq \lfloor \frac{n}{2} \rfloor + 1$ times in States **then**
17: $vote \leftarrow v$
18: **else**
19: $vote \leftarrow ?$
20: /* Round 2 */
21: Send (vote, $p, vote$) to all ▷$vote$ can be $0, 1$ or $?$
22: **wait until** receiving $\geq n - f$ phase-p vote messages
23: **if** a non-? value v appears $\geq f + 1$ times in votes **then**
24: **Return** FindReturnValue(v) ▷Termination
25: **else if** a non-? value v appears at least once in votes **then**
26: $state \leftarrow v$
27: **else**
28: $state \leftarrow$ CommonCoinFlip(p) ▷p-th coin flip
29: $p \leftarrow p + 1$ ▷Proceed to next phase

Algorithm 4 Weak-MEC: Helper Function

Procedure FindReturnValue(v)
1: **if** $v = 1$ **then**
2: Find ID that appears $\geq \lfloor f + k$ times
in Proposal's received in Phase 0
3: **Return** client operation q with ID
4: **else**
5: **Return** $\perp$ ▷return null value

- If the output is a write, then N_i persists the operation, along with its coded segment ($Code_w[i]$) to the disk for durability. Note that as mentioned earlier, Racos uses Reed-Solomon code and a coded segment can either be a data segment or a parity segment.
- If the output is a read *and* if N_i is client c's proxy (which is responsible for communicating with c), it needs to enter the next communication phase to fetch enough coded segments to decode and reconstruct the original data. It first finds ID_r – the ID of the most recent write on key_r (Line 13). If there is none, then the ID is 0. Then it contacts other nodes to obtain their coded segments. Upon receiving k segments, N_i uses RS library to decode and obtains $data_r$ – the data corresponds to the most recent written data, according to the ordering of the log (Line 16).
- If the output is a NULL operation, then N_i skips this slot.

Weak-MEC. Weak-MEC stands for Multi-Erasure Coding with weak guarantees. Specifically, Weak-MEC relies on the usage of Ben-Or's design and the common coin (the Randomized Binary Consensus Stage at Line 10 of Algorithm 3). In the Exchange Stage (Line 1), we add one more condition check to enable the fast-path.

At Line 4 in Algorithm 3, as long as nodes observe that $\lfloor n/2 \rfloor + f + 1$ proposals that contain the same ID, then they can immediately agree, *without* entering the RBC (randomized binary consensus) stage. This is because enough redundancy ensures that any node in the RBC stage must agree on the same value (for the particular slot in the log).

The Randomized Binary Consensus Stage at Line 10 in Algorithm 3 is similar to the one in Rabia, which relies on a common coin (Line 28) to break a tie. Given the RBC's outcome, the helper function is used to fetch the intended client operation or return the NULL operation. It is similar to the one in Rabia, except for a different threshold. Note that this component is "weak" in the sense that it can return a NULL operation. This design is inherited from Rabia, which is shown to lead to a more efficient consensus in general.

Spreader Component. The spreader is responsible for "spreading" the client operations to all the other nodes. Recall that Rabia and Racos are leaderless and any node can handle client operations. Therefore, we need a mechanism so that all nodes can observe all the operations (and then use the consensus component to figure out the ordering). Reads are handled similarly as in Rabia. The proxy node forwards the operation to others, and can push it to the priority queue. Upon receiving a forwarded operation (Line 26-27), a node can push the operation to the priority queue (Line 28).

Spreading write operations is more complicated, since this involves the erasure coding part. In Racos, node N_i first encodes the received data and obtains an array for coded segments using the encoding function. $Codes_w[k]$ corresponds to the coded segment that should be handled by node N_k. Node N_i then sends each node a corresponding segment and waits for $n - f$ acknowledgements. A deliberate design here is that nodes do not directly add a received write operation into the priority queue. Instead, the write is added to a buffer buf first. The operation is only added to the priority queue, *only if* a node receives a ready message (Line 24-25) or a node learns that at least $n - f$ nodes received corresponding segments (Line 20-22). This design choice ensures the decodability and in fact reduces the chance of conflicting operations, as discussed in Section 4.1.

Quorum Read. Algorithm 2 presents our quorum-based approach of handling the reads. If Quorum Read is enabled, then Algorithm 2 is used, which replaces Line 11-17 and Line 26-27. That is, in this case, the log contains only write operations. This leads to reduced chance of conflicting proposals.

Our approach is inspired by prior approaches like ABD register simulation [1], Gryff [5], Paxos Quorum Read [10]. However, our design is slightly simpler, because of the guarantees of the Weak-MEC and our design of the spreader. For example, we do not need to handle the case of leader failure as in Paxos Quorum Read or Gryff. Our quorum read consists of two phases:

- In the first phase, node N_i contacts a quorum of nodes to obtain the slot index $slot_r$ that contains the most recent write on the same key. The index $slot_r$ is calculated as the largest received slot index. This is necessary, as nodes might be at a different slot due to asynchrony. Line 8-9 implement the corresponding message handler.
- In the second phase, the proxy node N_i collects k coded segments to reconstruct the data using Decode, and return the data to the client. Line 10-11 implement the corresponding message handler.

4.3 Racos: Correctness and Analysis

4.3.1 Correctness. We show that given $n \geq 2f + k$, Racos is correct given that up to f nodes may crash. For brevity, we present only proof sketches below.

Theorem 1. *Racos satisfies liveness (in the probabilistic sense). In other words, all the operations complete with probability 1.*

Proof Sketch. The liveness follows from two observations:

- Weak-MEC satisfies the probabilistic termination, i.e., it terminates with probability 1. This is because we adopt Rabia's design (for RBC stage).

- All the thresholds used in Racos, except for Line 4 in Algorithm 3, do not go beyond $n - f$. Therefore, it is non-blocking, i.e., as long as $n-f$ nodes are responsive, then the algorithm can make progress. Note that Line 4 is a condition for fast-path, which does not affect termination.

□

Theorem 2. *Racos satisfies agreement. In other words, each slot contains the coded segment that corresponds to exactly the same ID.*

Proof Sketch. This again is similar to the proof for Rabia. The only exception is that we need to consider the case when some node takes the fast-path at Line 4 in Algorithm 3.

Suppose that is the case. We need to show that all nodes will eventually pass 1 to the Weak-MEC helper. Since some node executes Line 4 with an operation ID, this means that other nodes must have received at least $(\lfloor \frac{n}{2} \rfloor + f + 1) - f$ proposals with ID. This is because that due to asynchrony and the $n - f$ threshold used at Line 2, any pair of nodes can differ by at most $2f$ in received messages. Therefore, any other node will pass the condition at Line 6, update $state$ to 1. Consequently, all the $state$'s exchanged in the RBC stage is 1, leading to the case that every node that is not terminated at Line 4 will terminate at Line 24. This completes the proof. □

Theorem 3. *In Racos any acknowledged write is decodable. That is, there exists at least k coded segments in the system, for any completed write. A write is completed if Line 10 in Algorithm 1 is executed at some node.*

Proof Sketch. This follows from our design of the spreader component. In particular, due to the check at Line 20 in Algorithm 1. This implies that any write operation that is pushed into the priority queue must have its coded segments received by at least $n - f$ nodes. Since we assume $n \geq 2f + k$, this means that even after f crashes, there are at least $n - f \geq (2f + k) - f = f + k$ coded segments that are stored at nodes that have not crashed yet.

This observation together with the design that only the writes inside the priority queue can be completed prove the theorem. □

Theorem 4. *Quorum Read satisfies linearizability.*

Proof Sketch. Since when Quorum Read is used, reads are not stored in the log. We need to prove the linearizability separately. To show the property, we need to prove two conditions [21].

First, any read returns the most recent write. This is due to Line 2–3 in Algorithm 2. Line 2 ensures that there is a quorum intersection, and Line 3 ensures that the largest slot corresponds to the most recent write.

Second, any subsequent reads that occur after a completed read R must *not* return a write that is "older" than the write returned by R. This is guaranteed by Line 5. After it, it is guaranteed that at least k nodes that have already observed the slot that contains the write. Since $k \geq f$ in erasure coding SMR, this ensures that any subsequent reads cannot return older writes due to the quorum intersection guarantee. □

4.3.2 Analysis.

I/O and Storage Cost. Each node stores exactly one equal-size coded segment. Therefore, in a k-node system that stores M units of data, Racos has storage cost and disk I/O = M/k. The network I/O is equal to $\frac{M}{n}(1 + \frac{n-1}{k}) + \frac{M(n-1)}{n}\frac{1}{k}$, because of the evenly distributed workload.

RTTs. We now consider the RTTs on the fast-path:

- *Racos without Quorum Read*: reads take 0.5 RTT to go through the spreader, and another 0.5 RTT (at Line 2 of Algorithm 3) to complete. This leads to 1 RTT for reads.
- *Racos with Quorum Read*: Reads take two phases, which translate to 2 RTTs. This is because each phase requires a communication with a quorum of nodes.
- *Writes*: Writes take longer. The spreader takes 1.5 RTTs (Send's at Line 19 and 21 of Algorithm 1 and the acknowledgement at Line 20.) Weak-MEC takes 0.5 RTT to complete on the fast-path (by exiting at condition at Line 4 in Algorithm 3). This sums to 2 RTTs for writes.

In comparison, Rabia takes 2 RTTs to complete both writes and reads on the fast path. (In Rabia, the two phases in RBC stage have to be executed at least once). Raft and Paxos (and other similar leader-based erasure coding SMR) take 1 RTT.

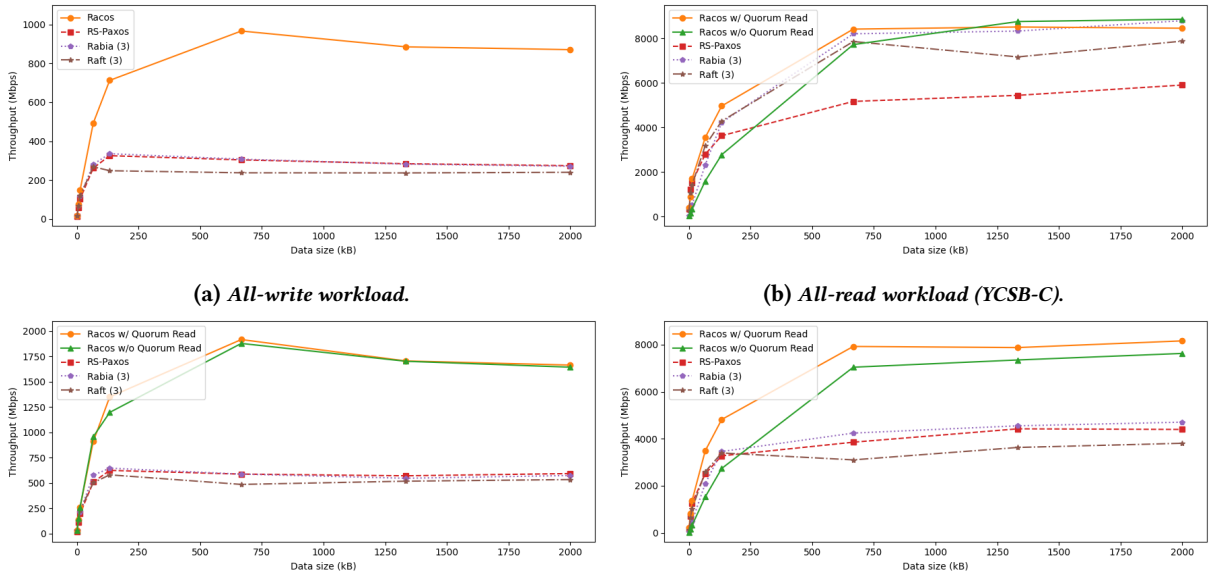
Following the analysis in [39], RBC stage takes on average 4 RTTs. Consequently, Racos takes 4.5 RTTs to complete a read and 5.5 RTTs to complete a write on average. If Quorum Read is enabled, reads *always* completed in 2 RTTs in Racos.

Even though each operation takes longer to complete than prior leader-based solutions, both Rabia and Racos can avoid the leader bottlenecks. Moreover, each node is able to serve operations, which leads to improved performance, as will be seen in our evaluation. As discussed in Section 4.1, compared to Rabia, Racos handles writes more efficiently, owing to our optimizations (i.e., a separated spreader to minimize the impact of non-pipelining and quorum reads to reduce conflicts.).

5 Evaluation

5.1 Implementation and Experiment Setup

We evaluate Racos by comparing it against three other systems Raft [38] (the original SMR used by etcd), Rabia [39]



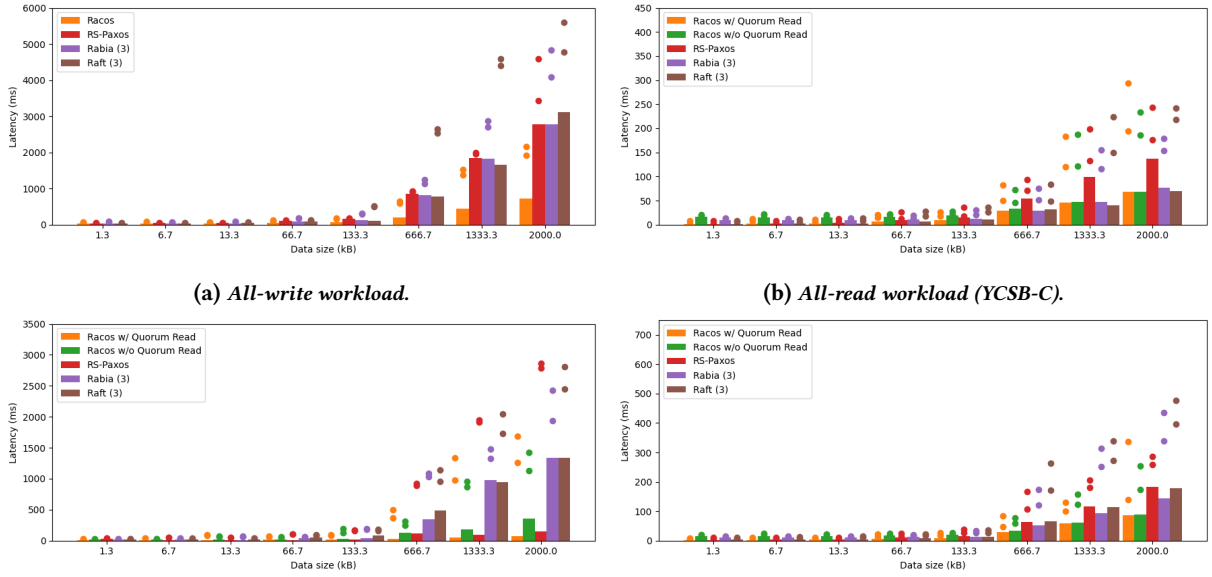
(a) ***All-write workload.***

(b) ***All-read workload (YCSB-C).***

(c) 50%***-write workload (YCSB-A).***

(d) 5%***-write workload (YCSB-B)***

Figure 1: ***Throughput vs. Varying Data Size.***



(a) ***All-write workload.***

(b) ***All-read workload (YCSB-C).***

(c) 50%***-write workload (YCSB-A).***

(d) 5%***-write workload (YCSB-B)***

Figure 2: ***Median latency (bar) and tail latencies (p95/p99) vs. Varying Data Size.***

(our base SMR system), and RS-Paxos [34]. RS-Paxos has the best performance in the common case, compared to other leader-based SMRs [25, 51, 54]. Therefore, we only use RS-Paxos as our competitor for the erasure coding solution.

We implement Rabia, RS-Paxos, and Racos into etcd, by following the implementation of Raft as closely as possible, e.g., using a similar pattern to exchange messages and accessing disks. Moreover, following the convention in [25, 34, 51, 54], the leader is assumed to hold the leader lease. Therefore, the reads can be completed without any communication by the leader. We do not assume lease or accurate clock synchronization in Raft, Rabia, or Racos.

We use machines on Emulab [52] for evaluation. Our experiment cluster consists of d430 nodes running Ubuntu 20.04.5 LTS. Each node is equipped with two 2.4 GHz 64-bit 8-Core CPUs, 64 GB RAM, and one 200 GB SSD. The network bandwidth between each pair of nodes is measured at 9.41 Gbits/s, using iperf. 1 RTT is around 0.125 ms. In our experiments, we have 5 to 13 server nodes that run etcd and 1 client node that runs Go-YCSB,[6] which is Go port of YCSB [13], to simulate realistic workloads that follow Zipfian distribution. Each run takes at least 30 seconds to minimize the impact of outliers.

By default, Racos and RS-Paxos have $n = 5$ and use $(3, 2)$-RS code, whereas Raft and Rabia have $n = 3$. All systems tolerate 1 crash fault. In Sections 5.6 and 5.7, we test different configurations.

Using FIO benchmark with 2MB-block and up to 4GB data, the measured write throughput of the SSD is around 1930 Mbps (megabits per second) and read throughput is around 3600 Mbps.

By default, the experiments use the YCSB default Zipfian skew parameter which represents highly skewed workloads (.99). In Section 5.9, we evaluate the case with lightly skewed workloads (.6).

5.2 Summary of Our Findings

To understand the performance and scalability of Racos, we aim at answering the following key questions:

- Does Racos improve maximum attainable throughput under varying data sizes and read/write ratios? (Section 5.3)
- Does Racos improve median latency and tail latency under varying data sizes and read/write ratios? (Section 5.4)
- Does Racos have smooth tradeoff between throughput and latency? (Section 5.5)
- Does Racos scale to a larger number of nodes? (Section 5.6)
- Does Racos handle smaller data with good performance? (Section 5.7)
- Does Racos behave similar in both highly and lightly skewed workloads? (Section 5.9)

To summarize our findings,

- In an all-write workload with data larger than 133.3 KBs, Racos has 2.13x-2.21x throughput improvement over the closest competitor (RS-Paxos and Rabia) when the systems saturate. In a balanced workload (50%-reads and 50%-writes), Racos has 1.81x-2.26x throughput improvement.
- In an all-write workload with data larger than 133.3 KBs, Racos improves median latency by 76.7%, compared to RS-Paxos. In a balanced workload, Racos improves by 55.8%, when "Quorum Reads" is enabled.
- In a balanced workload, Racos's median and p99 (99%-tile) latencies increase only slightly when the system starts to saturate, with a higher number of client threads. In a read-intensive workload (95%-reads and 5%-writes), the increase rates of median and p99 latencies are more smooth than other systems.
- In a balanced workload, Racos's throughput decreases only slightly from a 5-node cluster to a 13-node cluster (1.71% decrease). The coding is configured in a way that the 5-node cluster tolerates 1 fault, whereas the 13-node cluster tolerates 5 faults. Moreover, when systems are configured to tolerate 5 faults, Racos's throughput is 2.09x over the closest competitor, Rabia (in a 11-node cluster).
- Racos maintains performance advantages in both lightly and highly skewed workloads.

5.3 Maximum Attainable Throughput

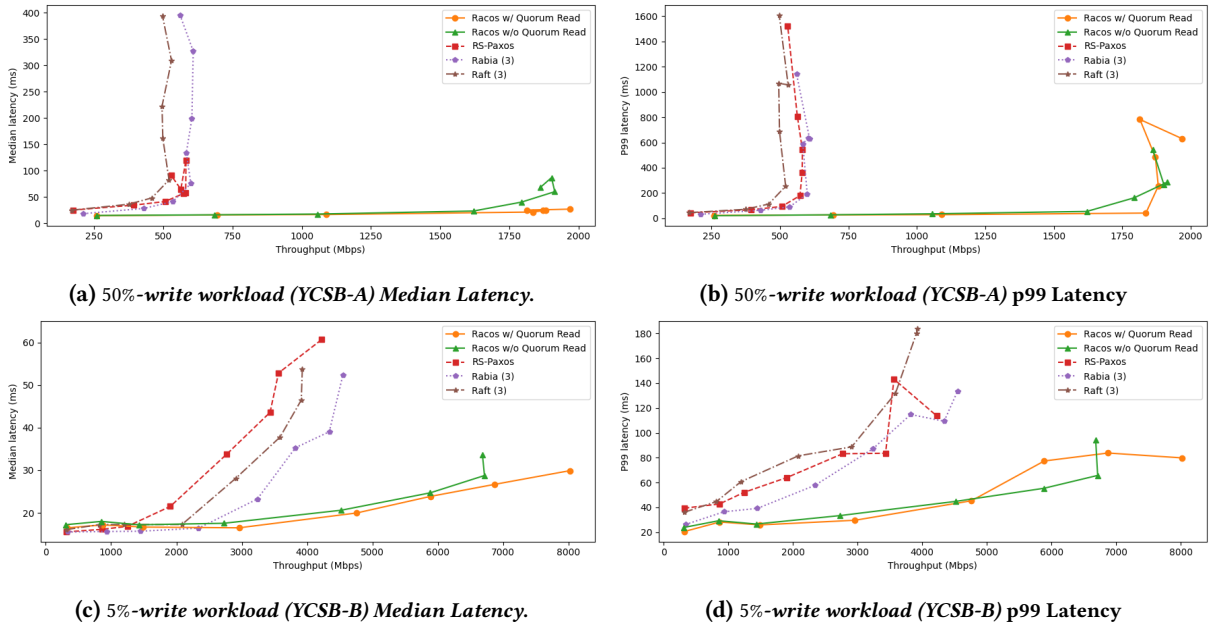
Figure 1 presents the maximum throughput (in Mbps) for each system under different workloads with different data sizes. We load each system gradually until saturation so that the throughput stays stable over a long period of time. The data size ranges from 1.3 KBs to 2000 KBs (2 MBs).[7]

We first measure a favorable setup for Racos, all-write workload. In this case, the disk I/O is the bottleneck, even for data as small as 1.3 KBs. Racos and RS-Paxos have $n = 5$ and use $(3, 2)$-RS code, whereas Raft and Rabia have $n = 3$. All systems tolerate 1 crash. Racos has 2.21x throughput improvement over RS-Paxos when the systems saturate, mainly because Racos writes less into disk (c.f., Table 1). Raft has the worst performance due to the saturated leader. Recall that its network I/O is higher than RS-Paxos. The YCSB-A which has a balanced workload shows a similar pattern. The only difference is that throughput is much higher, because the SSD can serve reads with a higher bandwidth.

In the case of all-read workload (YCSB-C workload), all systems, except for RS-Paxos, have similar performance at saturation. The main reason is that all systems, except for the RS-Paxos, can directly read default data from any node. For Raft, we follow the best practice, by connecting clients to all nodes, including the followers. This allows Raft's followers to serve reads in this case. As per the design of RS-Paxos, only the leader can serve reads. At larger data, Racos has a higher throughput without using Quorum Read. This is

[6] https://github.com/pingcap/go-ycsb/

[7] We control the data size by changing the fieldlength in YCSB, which controls the length of each field in its data structure. Then the YCSB client adds some overhead for metadata; hence, the size is not an integer.



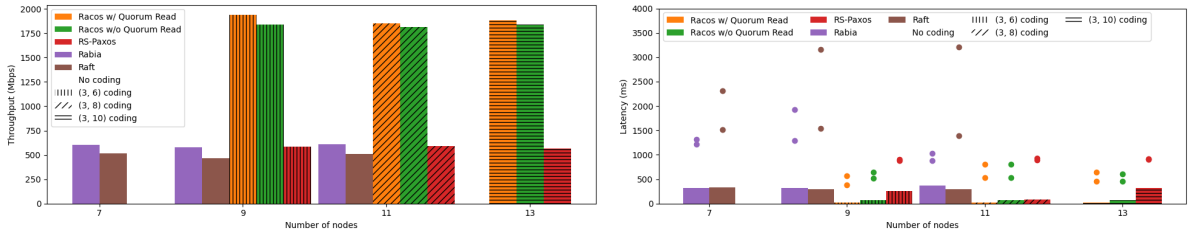
(a) 50%-*write workload (YCSB-A) Median Latency.*

(b) 50%-*write workload (YCSB-A)* **p99 Latency**

(c) 5%-*write workload (YCSB-B) Median Latency.*

(d) 5%-*write workload (YCSB-B)* **p99 Latency**

Figure 3: ***Throughput vs. Latency*** **(Data Size** = 666.7 **KBs).**



(a) 50%-*write workload (YCSB-A) Throughput.*

(b) 50%-*write workload (YCSB-A)* **Median and Tail Latency**

Figure 4: ***Scalability with 666.7 KBs.***

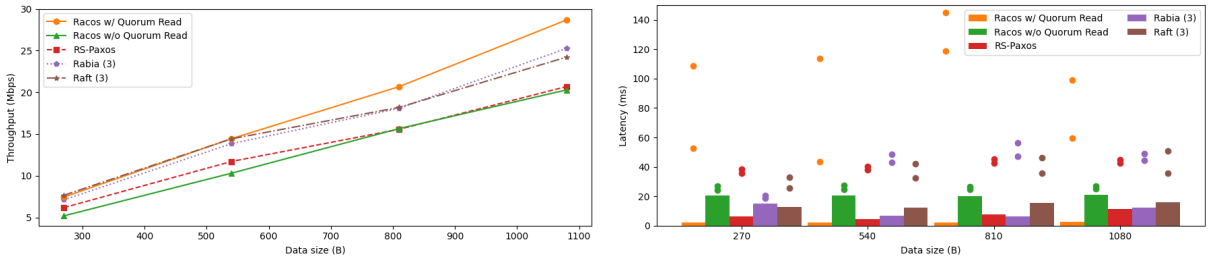
mainly because that Quorum Read requires more communication. Since all nodes can serve reads, the system throughput is higher than the read throughput of a single SSD.

However, such high performance is easily affected with a small increase in writes. With YCSB-B workload (5% writes), only Racos can maintain at a high throughput, roughly at 2x to the closest competitor. We can also observe that Quorum Read allows Racos to perform better when there are sufficiently number of writes.

5.4 Latency

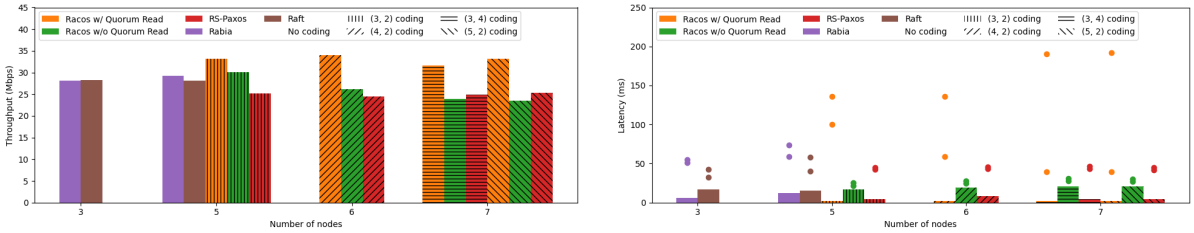
Figure 2 shows the median, 95%tile (p95), and 99%tile (p99) latencies (in ms) under the same workloads with varying data size. For smaller data, Racos and Rabia suffer from a higher tail latency due to the usage of randomized consensus and the lack of pipelining. However, as data size increases, the tail latency is lower than leader-based SMRs, due to the leaderless design and Racos's lower disk I/O cost.

Again, in all-read workload, Racos's Quorum Read introduces a higher tail latency due to its message complexity. However, similar to before, its median latency and p95 and p99 latencies improve when there is a sufficiently amount of writes. Interestingly, in the balanced workload (2c), the median latency of Racos without Quorum Read is larger than RS-Paxos, mainly because it assumes a leader lease and can serve reads without any communication. In comparison, in the case of Racos, the reads need to go through the consensus component and may introduce delay due to conflicts. This plot also demonstrates the effectiveness of Quorum Read, which reduces median latency by 55.8% to 76.7%.



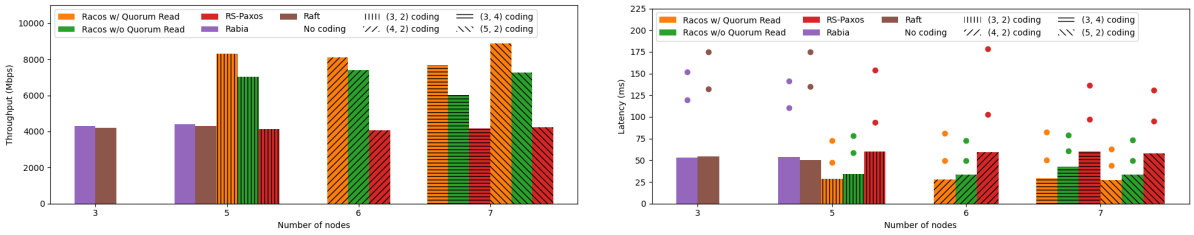
(a) ***Throughput vs. Data Size.*** **(b)** ***Median and Tail Latency***

Figure 5: ***50%-write workload (YCSB-A) with small data at Byte scale.***



(a) 50%***-write workload (YCSB-A) Throughput.*** **(b)** 50%***-write workload (YCSB-A)*** **Median and Tail Latency**

Figure 6: ***Scalability with 1.3 KBs.***



(a) 5%***-write workload (YCSB-B) Throughput.*** **(b)** 5%***-write workload (YCSB-B)*** **Median and Tail Latency**

Figure 7: ***Scalability with 666.7 KBs.***

5.5 Latency-Throughput Trade-off

Figure 3 presents the throughput-latency plot using a balanced workload (YCSB-A) ana a read-heavy workload (YCSB-B). We vary the number of client threads from 1 to 50 and collect the throughput and latencies at the particular load. In the balanced workload, Racos's median and tail latencies increase only slightly when the system starts to saturate. In the read-intensive workload, the increase rate of median and p99 latencies is more smooth than other systems (lower curve slope).

5.6 Scalability

Figure 4 presents the performance at a larger cluster size. The RS coding schemed is chosen in the way that a 9-node cluster erasure coding SMR tolerates 3 faults, a 11-node cluster tolerates 4 faults, and a 13-node cluster tolerates 5 faults. Full-copy systems (Raft and Rabia) follow the usual $n = 2f+1$ setup. The figure shows that even though Racos suffers a slightly performance degradation from a 5-node cluster (in Figure 1) to a 13-node cluster (1.71% decrease), it still has at least 2x throughput improvement over other systems.

5.7 Smaller Data and Varying Coding Scheme

While our targeted workloads are focused on data at KB and lower MB scale, we also evaluate the performance with small data at a few hundred bytes with a balanced workload. Figure 5 presents the performance. Racos without Quorum Read

has the lowest performance due to communication overhead and derailment from the fast-path due to conflicting proposals. Note that the operation-per-second is higher when handling smaller data, so the chance of conflict is higher. This demonstrates that Quorum Read is effective at lowering the conflicting proposals, making the performance comparable or better even with smaller data.

Figures 6 and 7 present the performance with different coding schemes and scalability with smaller data at 1.3 KBs and larger data at 666.7 KBs, respectively. Not surprisingly, Racos's performance advantage is not as great with smaller data. However, it demonstrates that Racos maintains stable performance with various coding schemes, showing that erasure coding is not a bottleneck when the storage systems persist data to disk.

At the scale of 1.3 KBs, Quorum Read introduces higher tail latency. Observe that in the GetSegment, nodes might need to catch up before sending their coded segments. However, with large data, both the median and tail latencies become the smallest when Quorum Read is enabled.

5.8 Unstable Network

Once concern for using randomized consensus is how well it performs under unstable network. This is because both Rabia and Racos rely on the natural ordering of messages to hit the fast path. Plus, with message delays and drops, the RBC component might take a longer time to complete. Therefore, we measure the performance with unstable network.

Figure 8 plots the performance with a balanced workload of 666.7KB-data, and Figure 9 considers a read-intensive workload. We use the Linux command *tc* to configure each communication link to drop its message at 0.01%. Recall that our targeted deployment is within a single datacenter which usually has a very stable network. Therefore, this message drop rate allows us to investigate an aggressive enough scenario.

Compared to the exact workload in Figure 1 when no message is dropped, throughput of all systems drop by a certain ratio. Since Racos requires more communication, it suffers more message drops, given the way we configure the per-link drop probability. However, even in this more adversarial case, Racos still maintains its performance advantage at around 0.66-1.5x improvement in throughput and around 25-33% decrease in median latency in both workloads.

5.9 Lightly Skewed Workloads

Figure 10 and Figure 11 evaluate the systems under lightly skewed workloads (with a Zipfian constant .6) using the same configurations as in Figure 1 and Figure 2, respectively. Racos still maintains its performance advantage (except for tail latencies in the all-read workload) across all configurations.

6 Related Work

There are tremendously amount of prior works on consensus and SMR. Here, we only focus on the most related systems that target either erasure coding or the deployment within a single datacenter. Many recent works improve the performance of Raft or Multi-Paxos, e.g., [8, 10, 11, 22, 26, 35, 41, 47, 50, 53]. All these systems and solutions are designed surrounding the idea of using a single leader to order operations and/or serve operations. Therefore, it is not clear how to integrate these techniques with erasure coding, while avoiding leader bottlenecks.

6.1 Erasure Coding SMRs

Erasure Coding SMRs. To our knowledge, RS-Paxos [34] is the first system that integrates SMR with erasure coding in the same design. As shown earlier, it suffers the leader bottlenecks, because it chooses to store the full copy of the data.

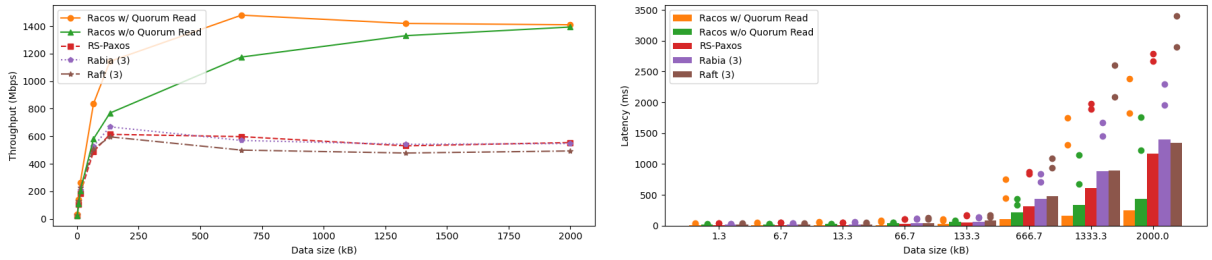
Subsequent works improve on various aspects [25, 49, 54]. CRaft [51] integrates erasure coding with Raft, and uses a failure detector to determine whether to use full-copy replication or erasure coding. It degrades to full-copy when the number of crashed nodes exceeds a certain threshold. HRaft [25] improves over CRaft, by reallocating coded segments to non-crashed nodes. That is, some nodes may need to store more than one coded segments. FlexRaft [54] take one step further by dynamically and adaptively change the coding scheme to better tradeoff storage cost and resilience. Compared to RS-Paxos, all these systems have the same or slightly worst common-case performance (due to extra coding overhead). Plus, it is not clear whether the assumption of *accurate* failure detector can maintain a stable performance in a real-world usage.

Pando [49] explores optimal storage-latency tradeoff in wide-area networks, by cleverly identifying a dynamic coding scheme and segment allocation scheme. Their technique is mostly useful when the communication cost is different on each link, which is common in wide-area networks. Racos does not consider the geo-replication workloads.

6.2 Distributed File Systems

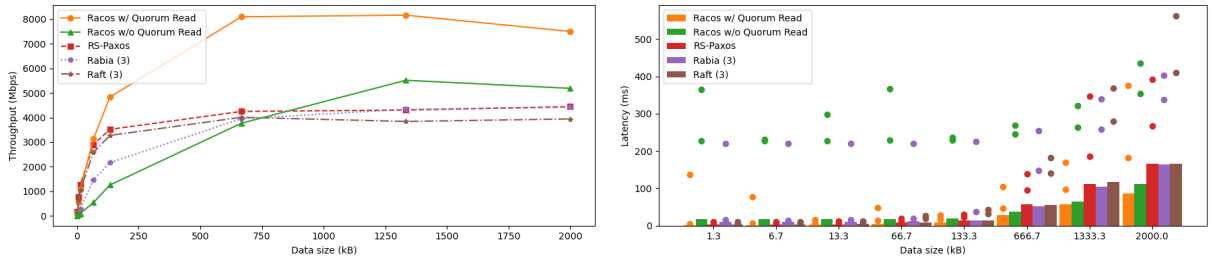
As mentioned earlier, erasure coding is adopted widely in many distributed file systems such as HDFS [20]. On a high-level, their fault-tolerance guarantee is different from the erasure coding SMR. Racos and other erasure coding SMR are always durable, as persistence-to-disk is on the critical path. In comparison, for file systems, durability is not always guaranteed. For example, HDFS does not currently guarantees persistence guarantees on . an erasure coded files.[8]

[8]HDFS Erasure Coding https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html



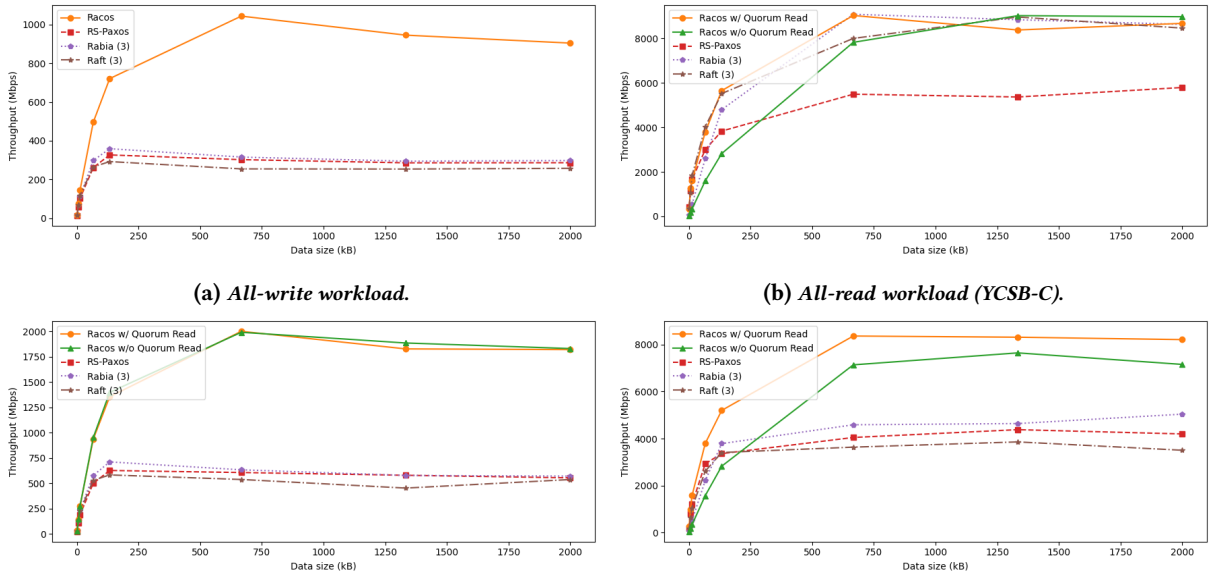
(a) ***Throughput vs. Data Size.*** **(b)** ***Median and Tail Latency***

Figure 8: ***50%-write workload (YCSB-A) in lossy network of 0.01% message drop per link.***



(a) ***Throughput vs. Data Size.*** **(b)** ***Median and Tail Latency***

Figure 9: ***5%-write workload (YCSB-B) in lossy network of 0.01% message drop per link.***

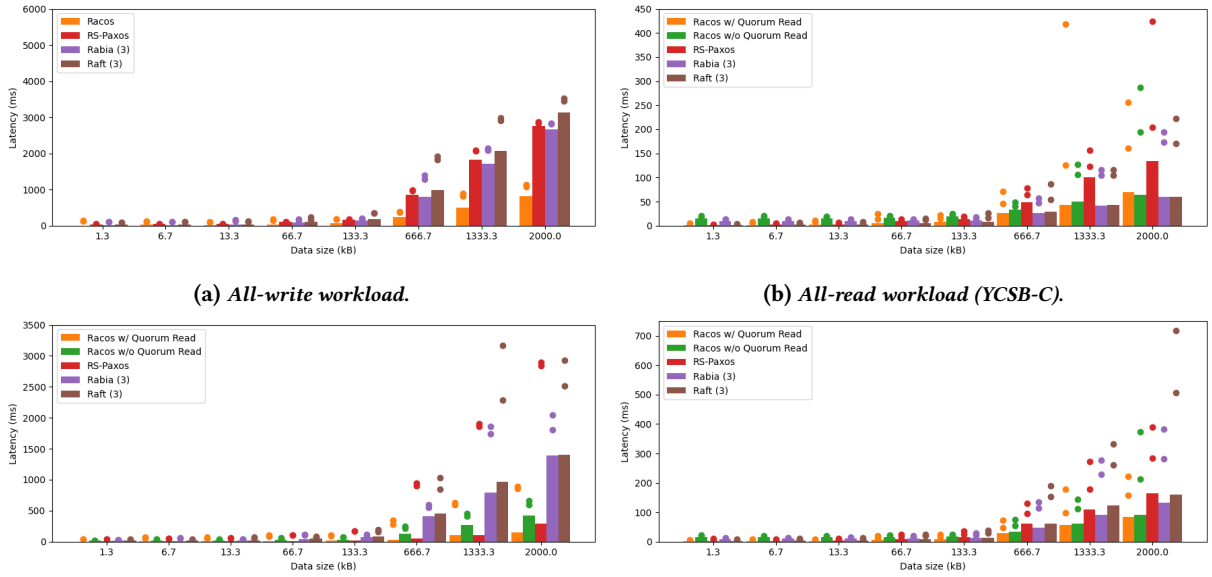


(a) ***All-write workload.*** **(b)** ***All-read workload (YCSB-C).***

(c) 50%***-write workload (YCSB-A).*** **(d)** 5%***-write workload (YCSB-B)***

Figure 10: ***Throughput vs. Varying Data Size (Lightly-Skewed).***

In short, these are different systems that make different design choices. distributed file systems support larger data and more complicated file structure by sacrificing certain persistence guarantees and semantics (e.g., write-once-read-many access model), whereas erasure coding SMR favors



(a) ***All-write workload.***

(b) ***All-read workload (YCSB-C).***

(c) 50%***-write workload (YCSB-A).***

(d) 5%***-write workload (YCSB-B)***

Figure 11: ***Median latency (bar) and tail latencies (p95/p99) vs. Varying Data Size (Lightly-Skewed).***

persistence, fault-tolerance, and richer access model (many-writer-many-reader). Therefore, one solution does not directly apply to another use scenario.

7 Summary

In this paper, we present our leaderless erasure coding SMR Racos. We implement Racos in etcd and use YCSB to demonstrate its performance advantage in both throughput and latency in wide variety of workloads.

References

[1] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. 1995. Sharing Memory Robustly in Message-Passing Systems. *J. ACM* 42, 1 (1995), 124–142. https://doi.org/10.1145/200836.200869

[2] Brian Beach. 2015. Backblaze Open-sources Reed-Solomon Erasure Coding Source Code. https://www.backblaze.com/blog/reed-solomon/.

[3] Michael Ben-Or. 1983. Another Advantage of Free Choice (Extended Abstract): Completely Asynchronous Agreement Protocols. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing* (Montreal, Quebec, Canada) *(PODC '83)*. Association for Computing Machinery, New York, NY, USA, 27–30. https://doi.org/10.1145/800221.806707

[4] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. 2011. Paxos Replicated State Machines as the Basis of a High-Performance Data Store. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*, David G. Andersen and Sylvia Ratnasamy (Eds.). USENIX Association. https://www.usenix.org/conference/nsdi11/paxos-replicated-state-machines-basis-high-performance-data-store

[5] Matthew Burke, Audrey Cheng, and Wyatt Lloyd. 2020. Gryff: Unifying Consensus and Shared Registers. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, Ranjita Bhagwan and George Porter (Eds.). USENIX Association, 591–617. https://www.usenix.org/conference/nsdi20/presentation/burke

[6] Michael Burrows. 2006. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, Brian N. Bershad and Jeffrey C. Mogul (Eds.). USENIX Association, 335–350. http://www.usenix.org/events/osdi06/tech/burrows.html

[7] 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. 2011. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 143–157. https://doi.org/10.1145/2043556.2043571

[8] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. 2018. PolarFS: An Ultra-low Latency and Failure Resilient Distributed File System for Shared Storage Cloud Database. *Proc. VLDB Endow.* 11, 12 (2018), 1849–1862. https://doi.org/10.14778/3229863.3229872

[9] Aleksey Charapko. 2022. Reading Group. Rabia. https://charap.co/reading-group-rabia-simplifying-state-machine-replication-through-randomization/, accessed Jan 2024.. In *Blog*.

[10] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. 2019. Linearizable Quorum Reads in Paxos. In *11th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2019, Renton, WA, USA, July 8-9, 2019*, Daniel Peek and Gala Yadgar (Eds.). USENIX Association. https://www.usenix.org/conference/hotstorage19/presentation/charapko

[11] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. 2021. PigPaxos: Devouring the Communication Bottlenecks in Distributed Consensus. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 235–247. https://doi.org/10.1145/3448016.3452834

[12] Yu Lin Chen, Shuai Mu, Jinyang Li, Cheng Huang, Jin Li, Aaron Ogus, and Douglas Phillips. 2017. Giza: Erasure Coding Objects across Global Data Centers. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, Dilma Da Silva and Bryan Ford (Eds.). USENIX Association, 539–551. https://www.usenix.org/conference/atc17/technical-sessions/presentation/chen-yu-lin

[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum (Eds.). ACM, 143–154. https://doi.org/10.1145/1807128.1807152

[14] 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. 2012. Spanner: Google's Globally-Distributed Database. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, Chandu Thekkath and Amin Vahdat (Eds.). USENIX Association, 251–264. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/corbett

[15] Jiaqing Du, Daniele Sciascia, Sameh Elnikety, Willy Zwaenepoel, and Fernando Pedone. 2014. Clock-RSM: Low-Latency Inter-datacenter State Machine Replication Using Loosely Synchronized Physical Clocks. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*. IEEE Computer Society, 343–354. https://doi.org/10.1109/DSN.2014.42

[16] Vitor Enes, Carlos Baquero, Alexey Gotsman, and Pierre Sutra. 2021. Efficient replication via timestamp stability. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar (Eds.). ACM, 178–193. https://doi.org/10.1145/3447786.3456236

[17] Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. 2020. State-machine replication for planet-scale systems. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer (Eds.). ACM, 24:1–24:15. https://doi.org/10.1145/3342195.3387543

[18] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (1985), 374–382.

[19] Vijay K. Garg. 2010. Implementing Fault-Tolerant Services Using State Machines: Beyond Replication. In *Distributed Computing, 24th International Symposium, DISC 2010, Cambridge, MA, USA, September 13-15, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6343)*, Nancy A. Lynch and Alexander A. Shvartsman (Eds.). Springer, 450–464. https://doi.org/10.1007/978-3-642-15763-9_44

[20] Apache Hadoop. 2023. HDFS Erasure Coding. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html.

[21] Maurice P Herlihy and Jeannette M Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492.

[22] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. 2016. Flexible Paxos: Quorum Intersection Revisited. In *20th International Conference on Principles of Distributed Systems, OPODIS 2016, December 13-16, 2016, Madrid, Spain (LIPIcs, Vol. 70)*, Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 25:1–25:14. https://doi.org/10.4230/LIPIcs.OPODIS.2016.25

[23] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. 2012. Erasure Coding in Windows Azure Storage. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, Gernot Heiser and Wilson C. Hsieh (Eds.). USENIX Association, 15–26. https://www.usenix.org/conference/atc12/technical-sessions/presentation/huang

[24] W. Cary Huffman and Vera Pless. 2003. *Fundamentals of Error-Correcting Codes*. Cambridge University Press. https://doi.org/10.1017/CBO9780511807077

[25] Yulei Jia, Guangping Xu, Chi Wan Sung, Salwa Mostafa, and Yulei Wu. 2022. HRaft: Adaptive Erasure Coded Data Maintenance for Consensus in Distributed Networks. In *2022 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2022, Lyon, France, May 30 - June 3, 2022*. IEEE, 1316–1326. https://doi.org/10.1109/IPDPS53621.2022.00130

[26] Marios Kogias and Edouard Bugnion. 2020. HovercRaft: achieving scalability and fault-tolerance for microsecond-scale datacenter services. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer (Eds.). ACM, 25:1–25:17. https://doi.org/10.1145/3342195.3387545

[27] Bradley C. Kuszmaul, Matteo Frigo, Justin Mazzola Paluska, and Alexander (Sasha) Sandler. 2019. Everyone Loves File: File Storage Service (FSS) in Oracle Cloud Infrastructure. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, Dahlia Malkhi and Dan Tsafrir (Eds.). USENIX Association, 15–32. https://www.usenix.org/conference/atc19/presentation/kuszmaul

[28] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *Operating Systems Review* 44, 2 (2010), 35–40. https://doi.org/10.1145/1773912.1773922

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

[30] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. 2008. Mencius: Building Efficient Replicated State Machine for WANs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, Richard Draves and Robbert van Renesse (Eds.). USENIX Association, 369–384. http://www.usenix.org/events/osdi08/tech/full_papers/mao/mao.pdf

[31] Parisa Jalili Marandi, Marco Primi, and Fernando Pedone. 2011. High performance state-machine replication. In *Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2011, Hong Kong, China, June 27-30 2011*. IEEE Compute Society, 454–465. https://doi.org/10.1109/DSN.2011.5958258

[32] Mark Marchukov. 2017. LogDevice: a distributed data store for logs. https://engineering.fb.com/2017/08/31/core-infra/logdevice-a-distributed-data-store-for-logs/.

[33] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is more consensus in Egalitarian parliaments. In *ACM SIGOPS 24th*

Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013, Michael Kaminsky and Mike Dahlin (Eds.). ACM, 358–372. https://doi.org/10.1145/2517349.2517350

[34] Shuai Mu, Kang Chen, Yongwei Wu, and Weimin Zheng. 2014. When paxos meets erasure code: reduce network and storage cost in state machine replication. In *The 23rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'14, Vancouver, BC, Canada - June 23 - 27, 2014*, Beth Plale, Matei Ripeanu, Franck Cappello, and Dongyan Xu (Eds.). ACM, 61–72. https://doi.org/10.1145/2600212.2600218

[35] Khiem Ngo, Siddhartha Sen, and Wyatt Lloyd. 2020. Tolerating Slowdowns in Replicated State Machines using Copilots. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 583–598. https://www.usenix.org/conference/osdi20/presentation/ngo

[36] 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. 2013. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, Nick Feamster and Jeffrey C. Mogul (Eds.). USENIX Association, 385–398. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala

[37] Geoffrey Noer and David Petrie Moulton. 2021. How Cloud Storage delivers 11 nines of durability—and how you can help. https://cloud.google.com/blog/products/storage-data-transfer/understanding-cloud-storage-11-9s-durability-target.

[38] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Philadelphia, PA) *(USENIX ATC'14)*. USENIX Association, USA, 305–320.

[39] 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 *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 472–487. https://doi.org/10.1145/3477132.3483582

[40] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva Shankar P., Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, Christian Preseau, Pratap Singh, Kestutis Patiejunas, J. R. Tipton, Ethan Katz-Bassett, and Wyatt Lloyd. 2021. Facebook's Tectonic Filesystem: Efficiency from Exascale. In *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, Marcos K. Aguilera and Gala Yadgar (Eds.). USENIX Association, 217–231. https://www.usenix.org/conference/fast21/presentation/pan

[41] Seo Jin Park and John K. Ousterhout. 2019. Exploiting Commutativity For Practical Fast Replication. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, Jay R. Lorch and Minlan Yu (Eds.). USENIX Association, 47–64. https://www.usenix.org/conference/nsdi19/presentation/park

[42] Tuanir França Rezende and Pierre Sutra. 2020. Leaderless State-Machine Replication: Specification, Properties, Limits. In *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference (LIPIcs, Vol. 179)*, Hagit Attiya (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 24:1–24:17. https://doi.org/10.4230/LIPIcs.DISC.2020.24

[43] Fred B. Schneider. 1990. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.* 22, 4 (1990), 299–319. https://doi.org/10.1145/98163.98167

[44] Pierre Sutra. 2020. On the correctness of Egalitarian Paxos. *Inf. Process. Lett.* 156 (2020), 105901. https://doi.org/10.1016/j.ipl.2019.105901

[45] Pasindu Tennage, Cristina Basescu, Lefteris Kokoris-Kogias, Ewa Syta, Philipp Jovanovic, Vero Estrada-Galiñanes, and Bryan Ford. 2023. QuePaxa: Escaping the tyranny of timeouts in consensus. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace (Eds.). ACM, 281–297. https://doi.org/10.1145/3600006.3613150

[46] Pasindu Tennage, Antoine Desjardins, and Lefteris Kokoris-Kogias. 2024. RACS and SADL: Towards Robust SMR in the Wide-Area Network. *CoRR* abs/2404.04183 (2024). https://doi.org/10.48550/ARXIV.2404.04183 arXiv:2404.04183

[47] Sarah Tollman, Seo Jin Park, and John K. Ousterhout. 2021. EPaxos Revisited. In *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*, James Mickens and Renata Teixeira (Eds.). USENIX Association, 613–632. https://www.usenix.org/conference/nsdi21/presentation/tollman

[48] Lewis Tseng, Neo Zhou, Cole Dumas, Tigran Bantikyan, and Roberto Palmieri. 2023. Distributed Multi-writer Multi-reader Atomic Register with Optimistically Fast Read and Write. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2023, Orlando, FL, USA, June 17-19, 2023*, Kunal Agrawal and Julian Shun (Eds.). ACM, 479–488. https://doi.org/10.1145/3558481.3591086

[49] Muhammed Uluyol, Anthony Huang, Ayush Goel, Mosharaf Chowdhury, and Harsha V. Madhyastha. 2020. Near-Optimal Latency Versus Cost Tradeoffs in Geo-Distributed Storage. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, Ranjita Bhagwan and George Porter (Eds.). USENIX Association, 157–180. https://www.usenix.org/conference/nsdi20/presentation/uluyol

[50] Yangyang Wang, Zikai Wang, Yunpeng Chai, and Xin Wang. 2021. Rethink the Linearizability Constraints of Raft for Distributed Key-Value Stores. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 1877–1882. https://doi.org/10.1109/ICDE51399.2021.00170

[51] Zizhong Wang, Tongliang Li, Haixia Wang, Airan Shao, Yunren Bai, Shangming Cai, Zihan Xu, and Dongsheng Wang. 2020. CRaft: An Erasure-coding-supported Version of Raft for Reducing Storage Cost and Network Cost. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, Sam H. Noh and Brent Welch (Eds.). USENIX Association, 297–308. https://www.usenix.org/conference/fast20/presentation/wang-zizhong

[52] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. 2002. An Integrated Experimental Environment for Distributed Systems and Networks. In *5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA, December 9-11, 2002*, David E. Culler and Peter Druschel (Eds.). USENIX Association. http://www.usenix.org/events/osdi02/tech/white.html

[53] Michael Whittaker, Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, Neil Giridharan, Joseph M. Hellerstein, Heidi Howard, Ion Stoica, and Adriana Szekeres. 2021. Scaling Replicated State Machines with Compartmentalization. *Proc. VLDB Endow.* 14, 11 (2021), 2203–2215. http://www.vldb.org/pvldb/vol14/p2203-whittaker.pdf

[54] Mi Zhang, Qihan Kang, and Patrick P. C. Lee. 2023. Minimizing Network and Storage Costs for Consensus with Flexible Erasure Coding. In *Proceedings of the 52nd International Conference on Parallel Processing, ICPP 2023, Salt Lake City, UT, USA, August 7-10, 2023*. ACM, 41–50. https://doi.org/10.1145/3605573.3605619