Distributed Multi-writer Multi-reader Atomic Register with Optimistically Fast Read and Write

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

Neo Zhou
zhouaea@bc.edu
Boston College
Boston, MA, USA

Cole Dumas
dumasca@bc.edu
Boston College
Boston, MA, USA

Tigran Bantikyan
tigranbantikyan@u.northwestern.edu
Northwestern
Evanston, IL, USA

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

ABSTRACT

A distributed multi-writer multi-reader (MWMR) atomic register is an important primitive that enables a wide range of distributed algorithms. Hence, improving its performance can have large-scale consequences. Since the seminal work of ABD emulation in the message-passing networks, many researchers study fast implementations of atomic registers under various conditions. "Fast" means that a read or a write can be completed with 1 round-trip time (RTT), by contacting a simple majority. In this work, we explore an atomic register with optimal resilience and "*optimistically fast*" read and write operations. That is, both operations can be fast if there is *no* concurrent write.

This paper has three contributions: (i) We present Gus, the emulation of an MWMR atomic register with optimal resilience and optimistically fast reads and writes when there are up to 5 nodes; (ii) We show that when there are > 5 nodes, it is *impossible* to emulate an MWMR atomic register with both properties; and (iii) We implement Gus in the framework of EPaxos and Gryff, and show that Gus provides lower tail latency than state-of-the-art systems such as EPaxos, Gryff, Giza, and Tempo under various workloads in the context of geo-replicated object storage systems.

CCS CONCEPTS

• **Theory of computation → Distributed algorithms**.

KEYWORDS

Register, Atomicity, Evaluation, Impossibility

ACM Reference Format:
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 '23), June 17–19, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3558481.3591086

SPAA '23, June 17–19, 2023, Orlando, FL, USA

ACM ISBN 978-1-4503-9545-8/23/06.
https://doi.org/10.1145/3558481.3591086

1 INTRODUCTION

Attiya, Bar-Noy, Dolev [7] present an emulation algorithm, namely ABD, that implements an atomic single-writer multi-reader register with optimal resilience in asynchronous message-passing networks when nodes may crash. ABD allows porting many known shared-memory algorithms to message-passing networks, such as multi-writer multi-reader (MWMR) registers, atomic snapshots, approximate consensus and randomized consensus.

The MWMR version of ABD [42] requires 2 RTT to complete a write operation, and 1 RTT to complete a read when there is no concurrent write. Subsequent works identify conditions so that reads [20, 34] and writes [23] can be fast. An operation is fast if it can always be completed in 1 round-trip time (RTT), by contacting a simple majority of nodes. Unfortunately, the conditions for fast writes are not generally applicable to practical systems as will be discussed in Section 2.2.

Dutta et al. [PODC '04] prove that implementing an atomic register with *both* fast writes and fast reads is impossible [20]. Recently, Huang et al. [PODC '20] identify more constraints in implementing fast writes or fast reads [34]. Motivated by these results, we ask: "*Can we do better for practical systems?*"

Motivation. Observe that object storage systems can be modeled as atomic registers. For real-world object storages, the typical workloads have two key characteristics [4, 14, 24]: (i) *Concurrency is rare, but possible*: in Microsoft OneDrive, only 0.5% of the writes occur within a 1 second interval; and (ii) *Object size and operation vary widely*: IBM Cloud Object Storage supports hosting service of web page, game, video and enterprise backups. In their testing benchmark [4], object size varies from 1 KB to 128 MB, and the ratio of write operations range from 5% to 90%.

These observations indicate that it is important to design an algorithm that handles various workloads efficiently, for practical object storages. To optimize for the common case, we are interested in "*optimistically fast*" operations, i.e., operations that are fast, when there is no concurrent write. The MWMR version of ABD [42] achieves optimistically fast reads, but not writes. Concretely, we answer the following question in this paper:

> Is it possible to implement an atomic register that supports both *optimistically fast* reads and writes?

Contribution: Theory. On the positive side, we present Gus, which implements an MWMR atomic register with optimal resilience and optimistically fast read and write operations when there are up to 5 nodes. To achieve optimistically fast operations, Gus combines two novel techniques: (i) *speculative timestamp*: a node optimistically uses locally known logical timestamp to enable 1-RTT fast path for writes (i.e., when writes commit in a single communication step), and (ii) *view exchange*: nodes exchange newly received timestamps to enable 1-RTT fast path for reads.

Considering that most production storage systems deploy 3- or 5-way geo-replication [16, 29, 49], we believe Gus is useful for practical settings, given its performance benefits. Furthermore, to address the scalability issue, we propose two solutions with different trade-offs between latency (in terms of RTTs) and resilience.

Furthermore, we formally prove that scalability is fundamentally limited. We show that when there are > 5 nodes, it is *impossible* to emulate an optimally resilient atomic register that supports optimistically fast reads and writes. This impossibility implies that Gus is optimal with this aspect.

Contribution: Systems and Experiments. We experimentally evaluate how the property of optimistically fast operations behave in object storages, as it is difficult to quantify how concurrent operations affect the performance in theory. Practical systems often use a consensus-based approach to implement an object storage. Hence, we compare Gus with state-of-art consensus-based systems EPaxos [44], Gryff [12],Giza [14], and Tempo [21].

We implemented Gus in the framework of EPaxos [44] and Gryff [12] to make a fair comparison. Furthermore, in the same framework, we implemented our version of Giza [14] (source code not available). Gus outperforms these competitors in both throughput and latency, which demonstrates practical performance benefits under a wide range of workloads. Under various settings with three nodes, Gus has better tail latency than both Gryff and EPaxos. Compared to Gryff, 5%-18% of Gus's reads are faster, and ≥95% of writes improve latency by up to 50%. Gus also has 0.5x-4.5x maximum throughput than both Gryff and EPaxos in the case of write-intensive geo-replication workload. With 9 nodes, Gus's tail latency for reads has 12.5% improvement over Tempo's [21].

2 PRELIMINARIES AND RELATED WORK

2.1 System Model

We consider an asynchronous message-passing network consisting of n nodes, where $n \leq 5$. Section 5.1 presents solutions to scale Gus beyond 5 nodes with different trade-offs. At most f of the nodes may crash at any point of time. Gus ensures safety and liveness as long as $n \geq 2f + 1$. Messages could be arbitrarily delayed, but messages between any pair of fault-free nodes are delivered eventually.

Following the convention of the literature [7, 8, 41], we assume that each node has client threads (reader thread or writer thread) and a server thread. In practical systems, this model captures co-located clients – a client C is co-located with a sever R if the message delivery latency between C and R is much less than the minimum latency between C and other servers. Clients running the applications (e.g., web hosting or backup service) can be considered co-located with a server in the same data center.

Linearizability. Gus achieves linearizability [32]. That is, there exists a total ordering of operations O such that (i) operations appear to execute in the order of O; (ii) O is consistent with register semantics, i.e., any read must return the value of the most recent write in O; and (iii) O satisfies the real-time ordering between operations, i.e., if operation o_a completes before the invocation of operation o_b, then o_b should appear after o_a in O.

2.2 Related Work

This section discusses the closely related theory works. We defer the comparison between Gus and relevant practical systems to Section 6. These systems (e.g., [12, 14, 21, 44]) are based on some form of consensus and provide liveness only in partially asynchronous networks, whereas Gus uses quorum and ensures both safety and liveness in asynchronous networks.

The ABD algorithm by Attiya, Bar-Noy, Dolev [7] is the first implementation of atomic single-writer multi-reader (SWMR) register in asynchronous networks with $n \geq 2f + 1$. ABD requires 1 RTT for writes and 2 RTT for reads. Lynch and Shvartsman [42] later extend the algorithm to the multi-writer multi-reader version, which takes 2 RTT for writes and 2 RTT for reads. These two versions of ABD support a simple optimization to make reads *optimistically fast*, i.e., 1 RTT reads when there is *no* concurrent write.

Subsequent works [20, 23, 34] study fast operations which complete in 1 RTT. The algorithm in [23] supports fast writes only when there are at most $n - 1$ writer clients. In practical geo-replication with n data centers, this condition implies that one data center cannot serve any writes. The algorithms in [20, 34] support fast reads, but require $n = O(fn_R)$, where n_R is the number of readers.

Prior works identify several impossibilities. Dutta et al. [20] show that in general it is impossible to have both fast writes and reads, when a single node may crash. Englert et al. [23] prove that to support fast writes, the number of writes cannot be more than $n - 1$ (which implies that their algorithm is optimal in this aspect). Huang et al. [34] derive two more impossibilities: (i) fast write is impossible if reads need to be completed in 2 RTT; and (ii) to have fast reads and 2-RTT writes, $\Omega(fn_R)$ is the lower bound on n.

Several works study other variations of the properties, e.g., semi-fast operations [28, 37], fast operations for Byzantine-tolerant SWMR registers [31], weak semi-fast operations [27], and fast operations for regular and safe registers [3, 30]. To the best of our knowledge, no prior work studies the feasibility of atomic registers with *optimistically fast* operations. Furthermore, our idea of speculative timestamp is new, which would be useful for future works that aim to achieve optimistically fast operations.

ABD Register [7, 42]. Most works on atomic registers in message-passing networks are inspired by ABD, including Gus. Hence, we briefly describe ABD before presenting our design. We describe ABD and Gus for a single register. Recall that linearizability is a local (or composable) property [32], i.e., the property holds for a set of objects, *if and only if* it holds for each individual object. Therefore, it is straightforward to compose instances of these protocols to obtain a linearizable system that supports multiple registers.

ABD associates a unique *tag* with a write and its value. Writes and values are ordered lexicographically by their tags. Formally, a tag is a tuple, (ts, id), consisting of two fields: (i) a logical timestamp

representing the (logical) time for the write; and (ii) the writer ID representing the writer client's identifier that invokes the write. For tag t, we use "$t.ts$" to denote the timestamp field, and "$t.id$" to denote the writer ID field. Two tags can be compared as follows:

Definition 1. *Tag t_1 is greater than tag t_2 if (i) $t_1.ts > t_2.ts$; or (ii) $t_1.ts = t_2.ts$ and $t_1.id > t_2.id$.*

Tag t_1 is equal to t_2 if $t_1.ts = t_2.ts$ and $t_1.id = t_2.id$.

Each node stores a value v and an associated tag t. ABD register requires two phases for both reads and writes. A read begins with the reader client obtaining the current tag and value from a quorum. The quorum is any simple majority of nodes. The reader then chooses the value associated with the maximum tag and propagates this maximum tag and value to all the nodes. Upon the acknowledgments from a quorum, the read is complete. The second phase, namely the "*write-back*" phase, can be omitted if all the tags from the first phase are identical, achieving *optimistically fast* reads.

A writer client w follows a similar two-phase procedure. It first obtains the maximum tag t_{max} from a quorum, and then constructs a new tag $t = (t_{max}.ts + 1, w)$. In the second phase, client w propagates t and its value to all nodes and waits for acknowledgments from a quorum. Since a writer needs to contact a quorum to obtain tag t (*writer-reads* design), ABD and relevant protocols [20, 34] require 2 RTT for the write operations, even if there is no concurrent operation. Our technique "speculative timestamp" and the focus on only 3 or 5 nodes allow us to skip this step optimistically.

3 GUS: DESIGN

3.1 Architecture and Protocol

Gus borrows tag and lexicographical ordering from ABD. A key challenge is to determine a tag for each write. Later we will show that even with a speculative timestamp, each write and its value still obtain a unique tag. As a result, we will often refer to a tag as the "version" of the register value.

Recall that each node has a writer, a reader and a server.[1] Writers and readers communicate with server threads at other nodes. For brevity, we will simply say writer/reader communicate with nodes. Readers exchange ⟨READ⟩ and ⟨ACK-READ⟩ messages, and writers exchange ⟨WRITE⟩, ⟨ACK-WRITE⟩, and ⟨COMMIT-WRITE⟩ messages. Background handlers of the server implement a set of event-driven functions that exchange ⟨UPDATE-VIEW⟩ messages with other nodes and update local variables.

Node States. Each node R_i maintains three states:

- $Storage_i$ is a set of tuples $(tag, value)$, which stores all the versions of the register value, where each version has a unique tag;
- tag_i represents the largest known tag associated with the value in $Storage_i$; and
- $View_i$ is a vector that keeps track of each node's view. View of a node R_i is defined as a set of tags that R_i has known so far. By design, $View_i[i]$ contains the tags associated to all the values in $Storage_i$. Condition SafeToReturn presented later in Definition 2 shows how Gus uses $View_i$ to decide which version of the register value is safe to return, with respect to linearizability.

We assume any thread on the node can access these states. This assumption is typical in many practical systems, as clients are handled by client proxies that run on each node.

Techniques and Challenges. Gus has two novel techniques:

- *Speculative timestamp*: Writer opportunistically uses the local tag tag_i as the tag for the value it intends to write.
- *View exchange*: Each node propagates to all the other nodes whenever it has learned a new tag. Each node i uses $View_i$ to keep track of this information.

Speculative timestamp allows Gus to achieve 1 RTT when there is no concurrent write, and enters the second phase only when observing a concurrent write. *View exchange* allows nodes to collect up-to-date information and to enable 1-RTT read when there is no concurrent write. In terms of protocol design, we need to address the following two technical challenges:

- No read can return stale value, even if the speculative timestamp is stale. A writer can observe a stale timestamp if the node that the writer is co-located with has *not* received the most recent writes from other nodes.
- No write operation can be associated with two tags. Essentially, the ordering of the operations is constructed using the tags; hence, if a write can be associated with multiple tags, the total ordering could be violated. We will formally define what "associated tags" mean after presenting the protocol.

Protocol Specification. Algorithm 1 specifies the steps that need to be followed by each node when $n = 3$. We defer the discussion of extension of $n = 4$ or 5 to Section 3.3.

Write operation: Writer i, which is co-located with node R_i, obtains tag (ts_i^{max}, i) by adding 1 to the timestamp of the largest tag known to node R_i (Line 2). It then propagates the value along with this new tag to all the nodes and waits until receiving an acknowledgement from a quorum of nodes Q (Line 4). A quorum used in Gus is always a *simple majority*.

Fast Path: Writer can then detect whether there is a concurrent write by comparing (ts_i^{max}, i) with the tag received from Q (Line 6). If there is no concurrent write operation, then i's write is on the fast path (Line 7). Client is notified that the write is completed at this point. The writer proceeds to asynchronous bookkeeping steps, including updating tag (Line 12), storage (Line 13), and view (Line 14). All these steps can be done asynchronously, because after Line 7, it is guaranteed that enough nodes have already obtained the value with the correct tag.

Slow Path: Only if the writer detects a concurrent write, it needs to obtain and update the correct logical timestamp. It first constructs the logical timestamp by finding the largest timestamp field in the received tags from Q and increasing it by 1 (Line 9). The writer then sends the commit message ⟨COMMIT-WRITE⟩ to all the nodes to update the tag, and waits for acknowledgement from a quorum on the slow path. This is necessary to ensure that enough nodes have received the correct and updated tag. Note that ⟨COMMIT-WRITE⟩ message does *not* include the value field to save network bandwidth.

Background Handler for Writes: The server thread has event-driven handlers that run in the background to process incoming

[1] Nodes can support multiple writers and readers using proxies.

Algorithm 1 Gus for $n = 3$

Write($value$) invoked by Writer i:
1: // *Put phase: propagate data with speculative timestamp*
2: $ts_i^{max} \leftarrow tag_i.ts + 1$
3: **send** $\langle$WRITE, $(ts_i^{max}, i), value\rangle$ **to all**
4: **wait until** receiving $\langle$ACK-WRITE, $tag_j\rangle$ **from a quorum of nodes** Q
5: // *Update phase: update tag if necessary*
6: **if** $(ts_i^{max}, i) > tag_j$ for all $j \in Q$ **then**
7: commit write ▷fast-path for write
8: **else**
9: $ts_i^{max} \leftarrow \max_{j \in Q}\ tag_j.ts + 1$
10: **send** $\langle$COMMIT-WRITE, $(ts_i^{max}, i)\rangle$ **to all**
11: **wait until** receiving $\langle$ACK-COMMIT, $(ts_i^{max}, i)\rangle$ from a quorum
12: $tag_i.time \leftarrow ts_i^{max}$; $tag_i.id \leftarrow i$
13: $Storage_i \leftarrow Storage_i \cup \{(tag_i, value)\}$
14: $View_i[i] \leftarrow View_i[i] \cup \{(tag_i, i)\}$
15: commit write, if not have already done so ▷slow-path for write

Read() invoked by Reader i:
16: **send** $\langle$READ, $i\rangle$ **to all**
17: **wait until** receiving $\langle$ACK-READ, $tag_j\rangle$ **from a quorum of nodes** Q
18: $tag^{max} \leftarrow$ largest tag_j received from all $j \in Q$
19: **wait until** Condition SafeToRead holds on $View_i, tag^{max}, value$
20: return $value$

Background Handlers at Node R_i:
21: **Upon receiving** $(\text{WRITE}, tag_j, value)$ **from writer** j:
22: **if** $tag_i < tag_j$ **then**
23: $Storage_i \leftarrow Storage_i \cup \{(tag_j, value)\}$
24: $tag_i \leftarrow tag_j$
25: $View_i[j] \leftarrow View_i[j] \cup \{tag_j\}$; $View_i[i] \leftarrow View_i[i] \cup \{tag_j\}$
26: **send** $\langle$UPDATE-VIEW, $tag_j\rangle$ **to all nodes**
27: **else**
28: $TmpStorage_i \leftarrow TmpStorage_i \cup \{(tag_j, value)\}$
29: **send** $\langle$ACK-WRITE, $tag_i\rangle$ **to writer** j
30: **Upon receiving** $(\text{COMMIT-WRITE}, tag_j)$ **from writer** j:
31: **if** j's write is in $TmpStorage_i$ **then**
32: $value \leftarrow$ value associated with j's most recent write in $TmpStorage_i$
33: $Storage_i \leftarrow Storage_i \cup \{(tag_j, value)\}$
34: **else**
35: Update $Storage_i$ to ensure j's write has tag tag_j
36: **if** $tag_i < tag_j$ **then**
37: $tag_i \leftarrow tag_j$
38: $View_i[j] \leftarrow View_i[j] \cup \{tag_j\}$; $View_i[i] \leftarrow View_i[i] \cup \{tag_j\}$
39: **send** $\langle$UPDATE-VIEW, $tag_j\rangle$ **to all nodes**
40: **send** $\langle$ACK-COMMIT, $tag_j\rangle$ **to writer** j
41: **Upon receiving** (READ, j) **from reader** j:
42: **send** $\langle$ACK-READ, $tag_i\rangle$ **to reader** j
43: **Upon receiving** $(\text{UPDATE-VIEW}, (ts, k))$ **from node** j:
44: $View_i[j] \leftarrow View_i[j] \cup \{(ts, k)\}$

messages. Upon receiving $\langle$WRITE$\rangle$ message from writer j, node R_i first checks the tag (ts, j). If it is larger than tag_i, then node R_i stores the value (Line 23), updates tag (Line 24) and view (Line 25), and notifies others that it has learned a new value (Line 26). Finally, R_i replies j with the acknowledgement (Line 29). If tag_i is larger, this means that writer j's tag may be stale, and j needs to update its speculative timestamp later. Hence, node R_i puts the value at a temporary storage $TmpStorage_i$ (Line 28) and replies j (Line 29).

Upon receiving $\langle$COMMIT-WRITE$\rangle$ message from writer j, node R_i moves the value from $TmpStorage_i$ to $Storage_i$ (Line 32, 33) if the write has been put in $TmpStorage_i$ before. Otherwise, R_i updates $Storage_i$ to make sure that j's write has the correct tag. The tag in $Storage_i$ could be stale if both R_i and j have not observed a previously completed write operation. Next, R_i proceeds with the steps similar to the previous handler: updates tag (Line 36, 37) and view (Line 38), and notifies others that it has learned a new value (Line 39). Finally, R_i sends acknowledgement to j (Line 40).

Technical Challenge 1: Due to asynchrony and failure, it is possible for a write to have a *stale speculative timestamp*. Consider the example in Figure 1, node R_3 has not observed the most recent write $write_1$; hence, its timestamp is still 1. Then, $write_2$, invoked by a writer co-located with R_3, has a stale tag because its speculative timestamp is less than the one included in a *completed* write operation $write_1$. Recall that to satisfy linearizability, a read that occurs after $write_2$ has to return the value of $write_2$, instead of $write_1$.

Gus achieves this by introducing the second phase to identify and update the correct timestamp, which equals to 3 in this example. After $write_1$ completes, R_1 and R_2 have timestamp 2; hence, after the first phase, $write_2$ learns the most recent timestamp from either node, and updates the correct tag in the second phase.

Read operation: In Gus, a reader can retrieve value from its co-located node. The only task is to figure out the value associated with the most recent tag, i.e., the version of the value that satisfies both real-time and total ordering constraints. Gus achieves this by

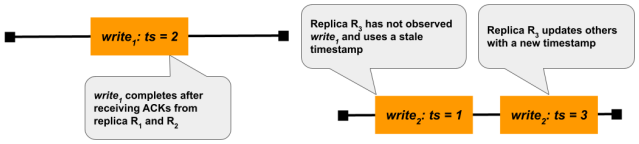


Figure 1: *Speculative Timestamp with* $n = 3$. Two ends denote the invocation and the completion time of each operation, respectively. Orange box denotes the timestamp (*ts*) field of each write's tag. $write_2$'s timestamp was stale initially.

first contacting a quorum of nodes to learn their most recent tag tag^{max} (Line 16 – 18), and using Condition SafeToRead, as per Definition 2, to obtain the return value (Line 19, 20).

Background Handler for Reads: Upon receiving $\langle$READ$\rangle$ message, node R_i returns its tag tag_i (Line 42), which is the largest known tag at R_i. Upon receiving $\langle$UPDATE-VIEW$\rangle$ message, node R_i updates the corresponding entry in $View_i$ (Line 44). By definition of $View_i$, adding (ts, k) to $View_i[j]$ means that node R_i learns that node R_j has added writer k's value associated with ts to $Storage_j$. Owing to the usage of speculative timestamp, it is possible that $View_i[j]$ has both (ts, k) and (ts', k) for the same k's write operation where $ts \neq ts'$. However, this does not affect the correctness, as we explain next how Gus addresses technical challenge 2.

Definition 2. *Condition SafeToRead is said to hold on $View_i$, tax^{max} and $value$ if there exists a $(tag, value)$ in $Storage_i$ where $tag \geq tag^{max}$, and there is a quorum Q_R s.t. (i) $i \in Q_R$; and (ii) for each $j \in Q_R$, $(tag, j) \in View_i[j]$.*

Intuitively, Condition SafeToRead finds a $(tag, value)$ pair in $Storage_i$ whose tag is larger than tag^{max} and $value$ is received by a quorum of nodes Q_R, including R_i. In other words, the condition ensures that the returned value is received by a read quorum Q_R, and its version tag is at least as recent as tag^{max}.

Figure 2 presents the fast and slow paths for reads. If $tag^{max} = tag_1$, then the condition must hold at that moment. If there is a

concurrent write (with a larger tag), then $tag^{max} > tag_1$. Thus, the reader at R_1 needs to wait for more messages – ⟨WRITE⟩ from R_5 and ⟨UPDATE-VIEW⟩ from two other nodes – to satisfy Condition SafeToRead. In the worst case, this takes 2 RTT.

Technical Challenge 2: With speculative timestamp, a write may have two tags (or timestamps). We say that a write is "*associated*" with a tag *tag* (or timestamp *ts*) if a read returns the value of a write with *tag* (or *ts*). In the example of Figure 1, we do not want $write_2$ to be associated with timestamp 1, i.e., no read should return the value of $write_2$ with timestamp 1. This is because that eventually $write_2$ will update its timestamp to 3, which means $write_2$ will be associated with two tags. Consequently, it is impossible to find a total ordering using associated tags that satisfies linearizability.

Condition SafeToRead is devised so that such an undesirable scenario can never occur. In Gus, a read returns *value* if a read quorum Q_R has received $(tag, value)$. In the aforementioned example, no read can return a value associated with timestamp 1 because R_1 and R_2 observe $ts = 1$ being stale, and R_3 updates *ts only after* $write_2$ is completed; hence, it is not possible to gather a read quorum. When $n = 3$, if a write observes a stale tag tag^{old}, then no read can return a value with tag^{old}. This is because at most one other node would consider tag^{old} as the most recent tag, which means that no read can obtain tag^{old} from a read quorum Q_R.

3.2 Correctness and Performance Analysis

We follow the proof structure in [7, 42], i.e., using tags to assign the order of the operations. The key difference is to prove that Gus addresses Technical Challenge 2 correctly – each write can only be associated with one tag. We prove the claim by formalizing the argument in Section 3.1. The complete proof is in [52].

Gus achieves optimistically fast operations, i.e., both writes and reads take 1 RTT if there is no concurrent write. Both operations take 2 RTT in the worst case, as shown in Figure 2. Message complexity for reads is the same as prior algorithms [7, 42], $O(n)$. For reads, we only count the messages on the fast path, since as shown in Figure 2, other messages for committing reads belong to writes. For writes, the message complexity is $O(n^2)$ due to ⟨UPDATE-VIEW⟩. Despite higher complexity, we find this acceptable in our target case because this design allows for using the fast path for reads. Moreover, for the case of object storage systems, ⟨UPDATE-VIEW⟩ only contains tag, not the data itself. Since typical data size is in the range of KBs, MBs or even more [4, 14, 24], the bit complexity and network bandwidth consumption of the overhead are negligible.

3.3 The Case of $n = 4$ or 5

Algorithm 1 does *not* work with $n > 3$ owing to *Technical Challenge 2* – a write could be associated with two tags when $n > 3$. Consider the example in Figure 3. Suppose $write_1$ is from a writer W_1 at R_1 and $write_2$ is from writer W_2 at R_2. Writer W_1 learns from R_2 that its speculative timestamp is stale due to the concurrent $write_2$. In the meantime, R_3, R_4, and R_5 have not observed $write_2$ and form a read quorum which allows a reader to read $write_1$ with a stale timestamp. After W_1 updates a new timestamp due to the notification from R_2, $write_1$ is associated with two tags.

To address this issue, more information needs to be included in ⟨ACK-WRITE⟩ message – if the highest tag is from R_j, then R_j needs

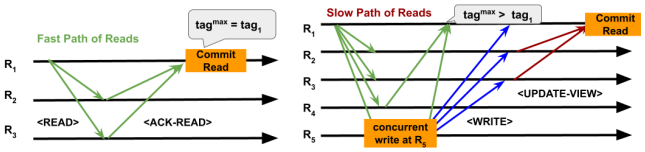


Figure 2: Fast/Slow Path of Reads. Green arrows represent ⟨READ⟩ and ⟨ACK-READ⟩, blue represent ⟨WRITE⟩, and red represent ⟨UPDATE-VIEW⟩. (On the right figure, not all messages are shown for brevity.)

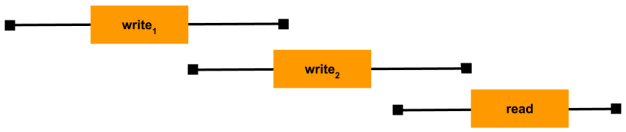


Figure 3: Example Execution. $write_2$ is concurrent with the other two operations.

to indicate whether a write is completed or not. In the earlier example, the second phase is not needed. Since $write_2$ has not completed yet (i.e., W_2 has not received a confirmation from a quorum), the writer W_1 does *not* need to update the tag, and can complete its write on the fast path. This does not violate linearizability, since by definition, two concurrent writes can appear in any order.

4 IMPOSSIBILITY

Theorem 1. *For $n > 5$ and $n = 2f + 1$, it's impossible to have an atomic register that supports optimistically fast writes and reads.*

Proof Sketch. The proof is based on an indistinguishability argument, which constructs several executions indistinguishable to nodes such that in one of the executions, a reader has to return a value that violates linearizability. All the executions we construct have no concurrent write; hence, the optimistically fast operations require all operations to complete in 1 RTT.

Consider $n = 7$ with nodes R_a to R_g. Since $f = 3$, the maximum quorum size to ensure liveness is 4. Now, consider the following executions such that the first write w_1 is invoked by a writer at node R_e and writes value x:

- E1: w_1 is completed with a write quorum $\{R_d, R_e, R_f, R_g\}$. All the messages from the write quorum to other nodes are delayed, except for the messages between R_d and R_a. At some time t after w_1 completes, reader at node R_a invokes a read r_1 that completes with a read quorum $\{R_a, R_b, R_c, R_d\}$ and returns w_1's value x.
- E2: Only node R_d receives w_1, because node R_e and its writer client crash during the write. The messages from R_e are all lost, because it has crashed. The messages from R_d to nodes other than node R_a are delayed. At time t, reader at node R_a invokes a read r_1 that completes with a read quorum $\{R_a, R_b, R_c, R_d\}$. Since E1 and E2 are indistinguishable from the perspective of node R_a and its reader, the read returns x.
- E3: Now, we construct E3 by extending E2. Right after the read r_1 completes, nodes R_a and R_d crash. This is allowed since $f = 3$. Furthermore, the messages from R_a and R_d to

all the other nodes are lost, because they have crashed. As a result, none of R_b, R_c, R_f, R_g learns the existence of w_1.
At some later time, a reader at R_b invokes another read r_2 that completes with a read quorum $\{R_b, R_c, R_f, R_g\}$, and returns a default value, violating linearizability.

It's straightforward to extend the argument to a larger n. □

Fundamentally, the impossibility is because that the quorum intersection is too small for a larger f. Due to the 1-RTT communication, readers or writers are *not* able to update all nodes in the read or write quorums. This is why we can defer messages in the proof. In the case of ABD, the read quorum of r_1 will learn the most recent value before r_1 completes because of the write-back. Consequently, the read quorum for r_2 would return x.

Note that in the construction above, we require 3 nodes to fail. This is why Gus works for $n \leq 5$. For example, when $n = 3$, the union of the reader and the quorum intersection is enough to ensure safety; hence, E3 is impossible and readers can learn the correct value that satisfies linearizability. To circumvent the impossibility, we either need to sacrifice optimistically fast operation or resilience.

5 PRACTICAL CONSIDERATIONS

5.1 Scalability

To increase scalability, we present two solutions for $n > 5$. The first increases 1 RTT for writes, which is suitable for serving larger objects because of its natural integration with erasure coding. The second increases the quorum size by focusing on the case of a smaller number of concurrent failures (relaxed resilience), a common case for modern geo-replicated systems [16, 21, 22]. For $n \leq 5$, these solutions are not needed, and therefore not applied.

Layered design by separating metadata and data: Inspired by Giza [14] and Layered Data Replication [25], we integrate a layered design with Gus, which separate the data and metadata paths into two layers. To read, a client first contacts the servers in the metadata layer to find the set of data servers that have the most recent data, and then reads the data from any of them. To write, a client first writes its data to a set of data servers, then update the metadata servers. Such a layered design allows the underlying data/metadata servers to optimize different workloads and features. Giza uses Azure object storage as the data server and Azure table as the core of metadata layer. Giza adopts Fast Paxos [39] to replicate the metadata (i.e., the version, the IDs and the locations of the object) to 3 or 5 metadata servers, whereas Gus uses Algorithm 1. In our implementation, we use Redis as the data server because of its high performance and support of durability.

As observed in [14, 53], to save storage and network cost, it is common to use erasure coding for the data layer. For larger objects, we adopt the $n = k + m$ Reed-Solomon code [35] – the value is divided into k data fragments, and the encoder generates m parity fragments. Each data server stores exactly one fragment. The object is durable as long as at most m node fails. With erasure coding, both writes and reads take 2 RTT on the fast path.

Increasing quorum sizes by lowering resilience: Concurrent failures in replication across datacenters are rare and transient [16, 21, 22]; hence, it is reasonable to focus on a smaller f with a larger n. Let Q_R and Q_W be the size of the read and write quorum, respectively. As long as they satisfy the following inequalities, Gus ensures safety: $2Q_W > n$ and $2n - 2Q_W - 1 < Q_R$
The first part ensures that any two write quorums intersect with each other, whereas the second part ensures that any write can only be associated with one tag (which can be argued similar as before). As long as a writer (or reader) can reach a write (or read) quorum, then its operation can be completed.

For read-intensive workloads, we can let Q_R be $\lfloor n/2 \rfloor + 1$. Then $Q_W > n - \frac{\lfloor n/2 \rfloor}{2} - 1$. For write-intensive workloads, we can lower write quorum size by increasing read quorum size accordingly. In other words, tolerating less failures allows Gus to explore a trade-off between quorum sizes and performance of different operations.

5.2 Optimizing Reads in Gus

We have two approaches to optimize reads in Gus. Consider the case of $n = 3$. Gus's read only needs 1 RTT with one simple change – piggybacking the value associated with the highest tag in ⟨ACK-READ⟩ at Line 42. Since any two nodes form a read quorum Q_R, upon receiving the value associated with tag^{max}, the reader can update $View$ and directly return the value, which must satisfy Condition SafeToReturn. The second optimization can be applied to the case when $n \leq 5$ and when a node serves several reader clients (a typical case in practical systems). Observe that read does not change the state at other nodes; hence, when there are multiple concurrent readers co-located in the same data center, then all the subsequent reads can "tag along" the first read *without* sending any messages.

6 EVALUATION

We evaluate Gus in practical settings. Our evaluation is focused on the case of geo-replicated object storages, because (i) atomic registers capture its semantic [12, 14]; and (ii) round-trip time matters the most for user-perceived latency in the case of geo-replication, as cross-datacenter latency can be in the order of 100+ms.

As discussed in Section 2.2, prior algorithms [20, 23, 34] with fast operations have limited practical usages due to their stringent conditions. Therefore, we compare Gus with consensus-based systems. Even though these systems only ensure liveness when the network is partially synchronous, they have high-performance in common cases. We first present related systems that are optimized for geo-replication, followed by our evaluation.

6.1 Related Work: Geo-replicated System

A comparison of Gus's features against state-of-the-art competitors is outlined in Table 1. To ensure a total ordering, storage systems often adopt the consensus-based approach. Most production systems [1, 2, 9, 11, 13, 16] rely on variants of Paxos [38, 40] or Raft [46] for agreeing on the order of client commands (or requests) and execute the commands following the agreed order. Unfortunately, these leader-based consensus protocols suffer long latency – 2 RTT (cross-datacenter message delay) – if the clients are not co-located with the leader data center.

Many recent systems [5, 21, 22, 43, 44, 47] propose a leaderless design to avoid the bottleneck at the leader and achieve optimistically fast operations.[2] EPaxos commits commands in 1 RTT when

[2]It is also called "optimal commit latency" in [44]; however, the term is typically used for consensus-based systems. Hence, we use a different term to avoid confusion.

	Latency		Fast-Quorum Size	Optimistically Fast Ops	Limitation
	Read	Write			
EPaxos [44]	1/2	1/2	$f + \lfloor (f+1)/2 \rfloor$	read/write	tail latency, dependency tracking
Gryff [12]	1/2	2/2	$f + \lfloor (f+1)/2 \rfloor$	read	write latency, throughput
Giza [14]	1/2	1/2	$\lfloor 3n/4 \rfloor + 1$	-	coordinator, large fast-quorum
Gus (this work)	1/2	1/2	$\lfloor n/2 \rfloor + 1$	read/write	all properties when $n = 3$ or 5

Table 1: Gus vs. related leaderless systems designed for geo-redundancy. The first number in the Latency column indicates the RTT in the common case (fast path), and the second is the RTT with contention. All of the systems tolerate f crashes with $n = 2f + 1$ nodes. Both EPaxos and Gryff support read-modify-write, whereas Giza and Gus do not. When n is beyond 5, some properties of Gus no longer hold.

there is no contention, and 2 RTT with contention. Unfortunately, EPaxos has worse tail latency than Paxos-based systems (up to 4x worse) [12] and may have a livelock in pathological cases [50]. This is mainly because EPaxos's fine-grained dependency tracking may chain dependency recursively, and the execution of some operations may be delayed in wide-area networks [50].

Gryff [12] reduces tail latency by unifying consensus and shared registers. Gryff implements an abstraction that provides read, write and read-modify-write (RMW) on a single object. On a high-level, it uses ABD register [7] to process reads and writes, and EPaxos [44] to process RMWs. While Gryff reduces p99 read latency compared to EPaxos, it always takes 2 RTT to complete a write; hence, it does not achieve optimistically fast writes and is not suitable for write-intensive workloads like game hosting or enterprise backup service that typically has around 90% of writes [4].

Giza uses Fast Paxos [39] to agree on the version for each operation, and needs only 1 RTT when there is no concurrent write. Two downsides of Giza are its reliance on the coordinator to order concurrent write operations and that its fast-quorum requires a super majority. Both affect tail latency, especially for the geo-replicated storage systems, because the clients need to wait for the nodes or the coordinator in the further datacenters.

Atlas [22] and Tempo [21] are two recent consensus-based systems that sacrifice resilience to optimize performance. Atlas uses dependency tracking; hence, suffers from long tail latency. Tempo develops a novel mechanism of using (logical) timestamps to determine when it is safe to execute a particular operation. Both systems have quorum size $\lfloor n/2 \rfloor + f$, which is optimal when $f = 1$. Atlas and Tempo do not distinguish between read and write quorums. Compared to them, Gus can be configured to have an optimal read quorum size, while having the write quorum size the same or greater by 1. Table 2 presents some examples. Gus's smaller read quorum not only allows a better read latency, but also ensures that reads can still complete, when $\geq \lfloor n/2 \rfloor + 1$ nodes are alive. For the case of $n = 11$, Tempo requires a quorum of 8, which equals to the write quorum of Gus. Reads can be served with a quorum of 6 in Gus. Later in Section 6.6, we will see how a smaller quorum size allows Gus to have better tail latency under practical workloads.

Other consensus-based systems achieve optimistically fast operations for both reads and writes, e.g., M^{2}Paxos [47], Caesar [5], and Mencius [43]. Each system performs well in certain cases. To support more general operations, e.g., transactions or RMW, they sacrifice high-performance under high skewed workload. Both EPaxos and Caesar use dependency tracking, which leads to high tail latency [50]. M^{2}Paxos requires a lock on an object; hence, not suitable for workloads with high contention. Mencius need information from all nodes.

		Gus		Atlas/Tempo
n	f	Q_R	Q_W	Q
7	2	4	5	5
9	2	5	7	6
11	3	6	8	8
13	3	7	10	9

Table 2: Read/write quorum size (Q_R/Q_R) in Gus, and quorum size (Q) in Atlas and Tempo, where f = number of tolerated concurrent failures.

6.2 Implementation and Experiment Setup

In our evaluation, we focus on tail latency, because it is well-known that user-perceived latency is correlated with the tail latency of the underlying storage systems [6, 18, 45, 48]. Our technical report [52] presents more evaluation results, including throughput comparison and the integration with erasure coding and layered design. We evaluate Gus against two categories of competitors: (i) those aiming/optimizing for fault-tolerant non-blocking MWMR registers (Gryff and Giza), and (ii) state-of-the-art consensus systems (EPaxos and Tempo) that are optimized for the same scenarios Gus is also optimized for.

For Gryff, we are essentially evaluating its ABD component (and Gryff's optimizations), as the workload consists of only reads and writes. For Giza, we only focus on the tail latency *without* any concurrent write. As documented in [14], its design is not optimized for concurrency. For scalability, we compare Gus with Tempo so that they tolerate the same number of concurrent failures.

Recent leader-based systems use techniques such as erasure coding [53, 54] and nil-externality [26] to further improve performance. We do not compare them because our evaluation is focused on leaderless systems. As shown in [12, 14, 44], leaderless systems have better performance in the common case and tail latency in the context of geo-replicated storages.

Implementation. We implemented Gus[3] and our version of Giza in Go using the framework of EPaxos and Gryff to ensure a fair comparison. For Tempo, we use the implementation in [21].

Clearly, even though in Algorithm 1 we focus on a single register (or object) for clarity, our implementation supports multiple objects and adopts the optimizations mentioned in Section 5. In order

[3] github.com/bc-computing/gus-automation

	CA	VA	IR	OR	JP
CA	0.2				
VA	72	0.2			
IR	151	88	0.2		
OR	59	93	145	0.2	
JP	113	162	220	121	0.2

Table 3: RTT (in ms) between VMs in emulated geographic regions [12]. For $n = 3$, we use VMs in CA, VA, and IR.

to do that, we include two extra fields in each message type – key *key* and sequence number *seq*. The key denotes the identifier of each object, and the sequence number is the operation index. This allows Gus to support multiple objects and also pipelining. We do *not* enable thrift optimization nor batching, because these optimizations generally increase the tail latency, by increasing the chance of conflicts [12, 44, 50].

In addition, we follow the same setup in [12, 44, 50] to separate node and client machines for best performance. Each node has several client proxies that handle requests from the respective client.

Testbed. We run our experiments on CloudLab [19] using m510 (Intel Xeon D-1548, 8 cores, 6 GB RAM) for node VMs and c6525-25g (AMD EPYC 7302P, 16 cores, 8 GB RAM) for the client VM. We adopt the same latency profile used in [12] – (i) $n = 3$: nodes in California (CA), Virginia (VA), and Ireland (IR); and (ii) $n = 5$: two more nodes in Oregon (OR) and Japan (JP). The latencies of the wide-area network are emulated using Linux's Traffic Control (tc) by adding delays to packets on all nodes with filters on different IPs. Table 3 shows the configured RTT between nodes in different regions. These numbers were chosen to represent typical RTT between the corresponding Amazon EC2 availability regions [12].

Experiment Setup. In all experiments, the clients run on one client VM, which adds no artificial latency to all the other node VMs. For most experiments, we use 16 closed-loop clients co-located with each node, again following [12]. This setup balances between capturing the effect of concurrent operations and avoiding saturating the system. This also allows us to isolate limitations of the hardware and software. We use different numbers of clients to stress the systems in the throughput experiment.

In our implementation, despite the fact that clients and servers are co-located, clients do not interact directly with the system's backend but pass through a proxy interface, which emulates an intermediate tier typically deployed in data centers for security and access control purposes. Therefore, the backend in our implementation supports remote clients by the usage of proxy.

For all experiments, we require the system to commit and execute client requests before responding to clients. Since as observed in [50], most applications depend on information or confirmation returned by an operation. For example, Redis and ZooKeeper [36] return results for both reads and writes.

Each experiment is run for 180 seconds, and we collect statistics in the middle of 150 seconds. In our experience, the statistics are quite stable during this period because of the removal of warm-up and cool-down. By default, each object is of 16B. While large objects are common in object storages [14, 24], 16B gives the best performance for EPaxos and Gryff, and in [6], Facebook reported the workload of using Memcached as a key-value store, where 40% of the data is less than 11B. Therefore, we mainly test 16B objects.

Following prior works [12, 44, 50], all the systems we test store the data in the main memory, except for two sets of experiments. This choice is reliable as long as the number of concurrent node failures is bounded. We are targeting redundancy across data centers, which are very rare to fail concurrently [14, 16, 22]. Moreover, there exist solutions that prevent data loss from crashed machines, e.g., persistent memory or disaggregated memory [17, 51, 55, 57]. For persistence, we write to a file using Go's OS package in Section 6.5. The m510 nodes are equipped with 256 GB NVMe flash storage.

Two operations are said to be conflicting with each other if they are targeting the same object (or same key) [12, 14, 44, 50]. Following the evaluation in [12, 21, 22, 44], we focus on the evaluations with various conflict rates. A conflict rate p denotes that a client chooses the same key with a probability p, and some unique key otherwise. Workload with a Zipfian distribution [50] shows a similar pattern.

6.3 Summary of Our Findings

To understand whether Gus performs well under various settings, we aim at answering the following questions:

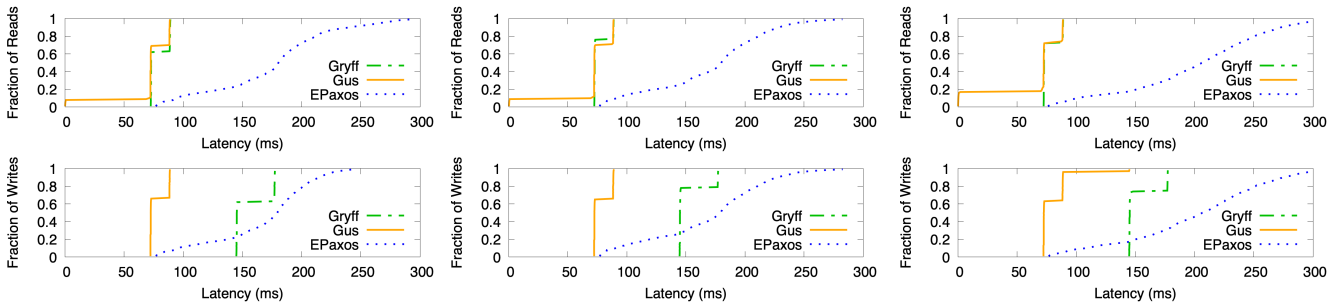
- Does Gus reduce tail latency under various conflict rates and write ratios? (Section 6.4)
- How does persistence affect latency? (Section 6.5)
- How does Gus scale when tolerating a smaller number of concurrent failures? (Section 6.6)

To summarize our findings, under various conflict rates, Gus has better read and write tail latency than both Gryff and EPaxos. When $n = 3$, around 5%-18% of reads are faster than Gryff, even though both systems complete reads in 1 RTT. This demonstrates the effectiveness of our read optimization mentioned in Section 5. Gus's maximum throughput is 0.5x-4.5x greater than Gryff and EPaxos in the context of geo-replication with a write-intensive workload. The write ratio does not have a significant impact on throughput in Gus, whereas it impacts Gryff significantly because of its 2-RTT writes. Finally, Gus's reads are 12.5%-17% faster than Tempo because of the smaller read quorum size.

6.4 Tail Latency

The Case of $n = 3$. First, we examine the tail latency of Gus, with a focus on large-scale web hosting. Since the web hosting applications is usually read-heavy [6, 10, 15], we use the ratio of 94.5% read operations with various conflict rate. This write ratio is the same as the YCSB-B workload [15]. Figure 4 presents the cumulative distribution functions (CDF) for both read and write latency for clients from three regions (CA, VA, IR) for three different conflict rates with $n = 3$. Top row represents the CDF for reads and bottom row for writes.

1 RTT Reads for Gus and Gryff. Both Gus and Gryff complete reads in 1 RTT, as shown in the top row of Figure 4. ~66% of reads complete after 1-RTT of communication with the nearest quorum (a simple majority) between CA and VA, which has latency of 72ms. Clients in IR are closest to the nodes in IR and VA, so 33% of the reads complete in 1 RTT between IR and VA, which is 88ms.



(a) 2% Conflict (b) 10% Conflict (c) 25% Conflict

Figure 4: Latency CDF ($n = 3$, 94.5% reads, 5.5% writes). Both Gryff and Gus complete reads in 1 RTT when $n = 3$.

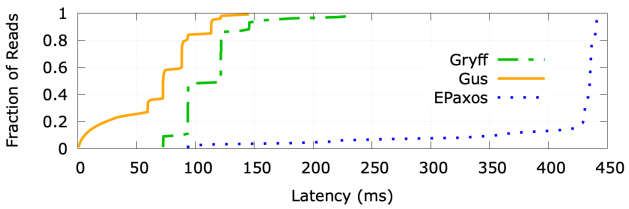


Figure 5: Latency CDF ($n = 5$, 50.5% writes, 25% conflicts). Gus's reads are faster than those of competitors due to the read optimization under high conflict rate and write ratio.

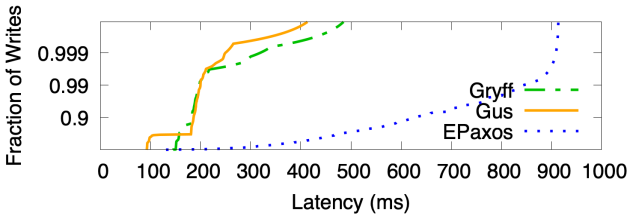


Figure 6: Log-scale Latency CDF with Persistent Writes ($n = 3$, 94.5% reads, 25% conflicts).

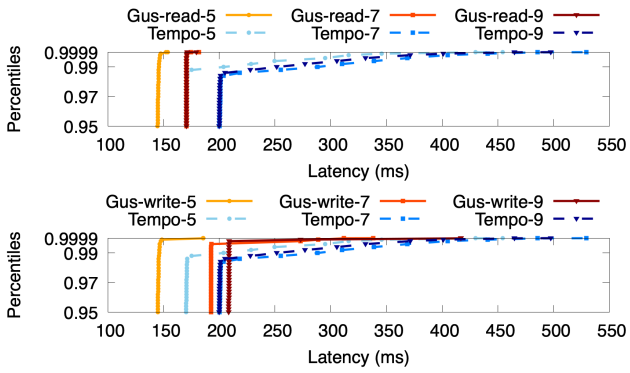


Figure 7: Log-scale Latency CDF with Scalability ($n = 5, 7, 9$; 94.5% reads; 2% conflicts).

Read Optimization of Gus. As mentioned in Section 5, Gus exploits the semantics of linearizable object storages to return reads without any communication when there are concurrent readers co-located within the same data center. Depending on when the concurrent reads are invoked, the latencies vary from 0.755ms to 72ms for clients in CA and VA.

Impact of Instant Execution. As identified in [12, 50], in EPaxos, some operations need to be delayed due to its dependency tracking, which results into a higher latency. In comparison, Gryff and Gus can execute an operation instantly.

Impact of Conflict Rate. For both Gryff and Gus, conflict rate does not affect latency significantly. This is because reads complete in 1 RTT, and writes always complete in 2 RTT in Gryff. With a higher conflict rate, Gus's reads have improved latency in the common case, owing to the read optimization. Higher conflict rate implies a higher chance for reads to tag along. With 25% conflict, Gus's writes occasionally need to take 2-RTT to complete.

The Case of $n = 5$. Figure 5 reports the cumulative distribution functions of the latency of reads and writes with $n = 5$. In this experiment, we use workload consisting of 49.5% reads and 50.5% writes with 25% conflicts. The write ratio follows from YCSB-A. Roughly equal amount of operations and the higher conflict rate allow us to observe the performance under concurrent operations.

6.5 Persistence After Crash

For persistence, we log state change to an SSD disk before sending acknowledgement. This experiment uses the same configuration as in Figure 4c. In EPaxos, nodes log synchronously to an SSD-backed file, whereas in Gryff and Gus, nodes log their state change only for incoming writes; hence, we only report the latency for writes in Figure 6. All the systems are I/O bound, but EPaxos's dependency tracking makes the tail latency increase by ~600ms, whereas Gryff and Gus increase by 280-300ms. Even with persistent writes, Gus still has better tail latency because of its 1-RTT fast path.

6.6 Scalability: Relaxed Resilience

In Figure 7, we compare Gus against Tempo [21] with $n = 5, 7, 9$. Both systems tolerate 2 concurrent failures in all three scenarios. To avoid cluttering the plot, we omit the results of EPaxos, Gryff, Flexible Paxos [33], and Atlas [22], because they generally have higher tail latency, as also observed in [21, 50].

Gus has better tail latency for reads because of its smaller read quorum (see Table 2). For example, when $n = 5$, Gus's fast path to the closet fast quorum takes 72-145ms and Tempo's takes 93-162ms. In general, Tempo has better latency for writes when $n = 9$,

because its quorum is 1 less than Gus's write quorum. Occasionally, Tempo needs to wait for timestamps to becomes stable to execute an operation. This is mainly the reason that Gus outperforms Tempo when we consider p99 or above latency for writes.

ACKNOWLEDGEMENT

Authors would like to thank the anonymous reviewers for the constructive comments. Authors would also like to thank Vitor Enes for helping the evaluation on Tempo. This material is based upon work partially supported by the National Science Foundation under Grant CNS-2045976 and CNS-1816487.

REFERENCES

[1] Cockroachdb. https://www.cockroachlabs.com/.
[2] etcd. https://etcd.io/.
[3] I. Abraham, G. Chockler, I. Keidar, and D. Malkhi. Byzantine disk paxos: optimal resilience with Byzantine shared memory. *Dist. Comp.*, 18(5):387–408, 2006.
[4] A. Anwar, Y. Cheng, A. Gupta, and A. R. Butt. Taming the cloud object storage with MOS. In *Proceedings of the 10th Parallel Data Storage Workshop, PDSW 2015*.
[5] B. Arun, S. Peluso, R. Palmieri, G. Losa, and B. Ravindran. Speeding up consensus by chasing fast decisions. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017*.
[6] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *ACM Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '12*.
[7] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, Jan. 1995.
[8] H. Attiya and J. Welch. *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. John Wiley & Sons, 2004.
[9] J. Baker et al. Megastore: Providing scalable, highly available storage for interactive services. In *Fifth Biennial Conference on Innovative Data Systems Research, CIDR 2011, Asilomar, CA, USA, January 9-12, 2011*.
[10] N. Bronson et al. TAO: facebook's distributed data store for the social graph. In *2013 USENIX Annual Technical Conference 2013*.
[11] M. Brooker, T. Chen, and F. Ping. Millions of tiny databases. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 463–478. USENIX Association, 2020.
[12] M. Burke, A. Cheng, and W. Lloyd. 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*.
[13] B. Calder et al. 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 2011.
[14] Y. L. Chen et al. Giza: Erasure coding objects across global data centers. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017*.
[15] B. F. Cooper et al. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010*.
[16] J. C. Corbett et al. Spanner: Google's globally-distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012*.
[17] Intel. Intel optane persistent memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html, 2021.
[18] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, 2013.
[19] D. Duplyakin et al. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.
[20] P. Dutta et al. How fast can a distributed atomic read be? In *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, PODC 2004*.
[21] V. Enes, C. Baquero, A. Gotsman, and P. Sutra. Efficient replication via timestamp stability. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, pages 178–193. ACM, 2021.
[22] V. Enes, C. Baquero, T. F. Rezende, A. Gotsman, M. Perrin, and P. Sutra. State-machine replication for planet-scale systems. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 24:1–24:15. ACM, 2020.
[23] B. Englert et al. On the efficiency of atomic multi-reader, multi-writer distributed memory. In *OPODIS*, 2009.
[24] O. Eytan et al. It's time to revisit LRU vs. FIFO. In *12th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2020*.
[25] R. Fan and N. A. Lynch. Efficient replication of large data objects. In *Distributed Computing, 17th International Conference, DISC 2003*.
[26] A. Ganesan, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Exploiting nil-externality for fast replicated storage. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles 2021*.
[27] C. Georgiou, N. C. Nicolaou, and A. A. Shvartsman. On the robustness of (semi) fast quorum-based implementations of atomic shared memory. In *Distributed Computing, 22nd International Symposium, DISC 2008*.
[28] C. Georgiou, N. C. Nicolaou, and A. A. Shvartsman. Fault-tolerant semifast implementations of atomic read/write registers. *J. Parallel Distributed Comput.*, 69(1):62–79, 2009.
[29] S. Ghemawat, H. Gobioff, and S. Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003*.
[30] R. Guerraoui and M. Vukolic. How fast can a very robust read be? In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing, PODC 2006, Denver, CO, USA, July 23-26, 2006*, pages 248–257. ACM, 2006.
[31] R. Guerraoui and M. Vukolic. Refined quorum systems. *Distributed Comput.*, 23(1):1–42, 2010.
[32] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.
[33] H. Howard, D. Malkhi, and A. Spiegelman. Flexible paxos: Quorum intersection revisited. In *20th International Conference on Principles of Distributed Systems, OPODIS 2016*.
[34] K. Huang, Y. Huang, and H. Wei. Fine-grained analysis on fast implementations of distributed multi-writer atomic registers. In *PODC '20: ACM Symposium on Principles of Distributed Computing 2020*.
[35] W. C. Huffman and V. Pless. *Fundamentals of Error-Correcting Codes*. Cambridge University Press, 2003.
[36] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *2010 USENIX Annual Technical Conference*.
[37] K. M. Konwar, S. Kumar, and L. Tseng. Semi-fast byzantine-tolerant shared register without reliable broadcast. In *Proceedings of the 40th IEEE International Conference on Distributed Computing Systems ICDCS 2020*. IEEE, 2020.
[38] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.
[39] L. Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.
[40] L. Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.
[41] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
[42] N. A. Lynch and A. A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *The Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing, 1997*.
[43] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building efficient replicated state machine for wans. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008*.
[44] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *24th Symposium on Operating Systems Principles, SOSP '13*.
[45] R. Nishtala et al. Scaling memcache at facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013*.
[46] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference*.
[47] S. Peluso, A. Turcu, R. Palmieri, G. Losa, and B. Ravindran. Making fast consensus generally faster. In *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016*.
[48] H. Scalability. Latency is everywhere and it costs you sales - how to crush it http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it, 2009, accessed 2021.
[49] D. B. Terry et al. Consistency-based service level agreements for cloud storage. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP 2013*.
[50] S. Tollman, S. J. Park, and J. K. Ousterhout. Epaxos revisited. In *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021*.
[51] S. Tsai, Y. Shan, and Y. Zhang. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020*.
[52] L. Tseng, N. Zhou, C. Dumas, T. Bantikyan, and R. Palmieri. Distributed multi-writer multi-reader atomic register with optimistically fast read and write, 2023. *CoRR*, abs/2304.09382, 2023.
[53] M. Uluyol, A. Huang, A. Goel, M. Chowdhury, and H. V. Madhyastha. Near-optimal latency versus cost tradeoffs in geo-distributed storage. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020*.
[54] Z. Wang, T. Li, H. Wang, A. Shao, Y. Bai, S. Cai, Z. Xu, and D. Wang. 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*.
[55] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies, FAST 2020*.
[56] H. Zare, V. R. Cadambe, B. Urgaonkar, C. Sharma, P. Soni, N. Alfares, and A. Merchant. Legostore: A linearizable geo-distributed store combining replication and erasure coding. *CoRR*, abs/2111.12009, 2021.
[57] M. Zhang, Y. Hua, P. Zuo, and L. Liu. FORD: Fast one-sided RDMA-based distributed transactions for disaggregated persistent memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*.