Asymmetric Linearizable Local Reads

Myles Thiessen
University of Toronto
Toronto, Canada
mthiessen@cs.toronto.edu

Guy Khazma
University of Toronto
Toronto, Canada
guykhazma@cs.toronto.edu

Sam Toueg
University of Toronto
Toronto, Canada
sam@cs.toronto.edu

Eyal de Lara
University of Toronto
Toronto, Canada
delara@cs.toronto.edu

ABSTRACT

Many linearizable local read algorithms have been proposed to minimize the read latency of strongly consistent distributed databases deployed in geo-distributed networks. These algorithms do so by enabling reads to be performed immediately against any process' copy of the database in the best case. However, as our analysis shows, worst-case read latency at every process with all existing algorithms is at least the network's relative diameter in terms of the maximum message delay minus a known lower bound on message delay between any two processes. We then show that by leveraging the asymmetric message delays of geo-distributed networks, worst-case read latency can be below the network's relative diameter at processes close to the leader or the network's center by presenting two new linearizable local read algorithms. Our experimental evaluation shows that these new algorithms reduce worst-case read latency by up to 50x compared to existing ones.

PVLDB Reference Format:
Myles Thiessen, Guy Khazma, Sam Toueg, and Eyal de Lara. Asymmetric Linearizable Local Reads. PVLDB, 18(8): 2427 - 2439, 2025.
doi:10.14778/3742728.3742738

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/myles-thiessen/pairwise.

1 INTRODUCTION

Linearizability is a desirable property for distributed databases because it provides the illusion that the database is run on a single machine that performs operations one at a time [17]. Applications built on top of these databases are more reliable and maintainable because developers do not need to reason about side effects that can arise from concurrent operations [16, 22].

To provide linearizability, databases often use a state machine replication algorithm such as Paxos [21] or Raft [27]. These algorithms totally order all operations by electing a distinguished process known as the *leader* to assign each operation a unique index number. They also ensure that the database remains available despite process crashes by replicating the entire database across multiple *followers*, i.e., processes other than the leader.

In geo-distributed networks, these algorithms take tens to hundreds of milliseconds to perform an operation because they synchronously communicate with distant datacenters. To avoid this communication when performing reads, the predominant operation in practice [11], many linearizable local read algorithms have been proposed to perform reads *without sending any messages* [5, 6, 9, 20, 26, 29]. These algorithms enable *all* processes to perform reads *immediately* against their local copy of the database in the best case. However, in the presence of concurrent database changes, reads may *block* for some time. This is to ensure that if a read r is performed against version V of the database, then every read that begins after r will be performed against version V or later.

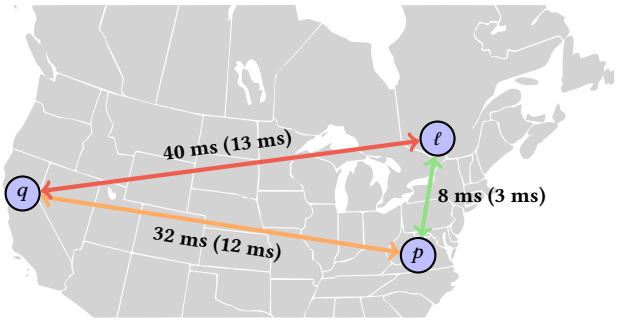


Figure 1: Average message delays (and speed of light propagation times) between three AWS regions.

In this paper, we first show that in periods where message delays are fixed and all processes are non-faulty[1], existing algorithms fall short in minimizing worst-case read blockage times when deployed on commodity hardware.[2] This is because their worst-case read blockage time at every follower is at least the *network's relative diameter*, i.e., the maximum message delay minus a known lower bound on message delay between any two processes. With state-of-the-art existing algorithms, this is because they schedule "events" using *globally synchronized clocks*, their worst-case read blockage time is the bounds on clock skew Δ, and it is known that without specialized clock synchronization hardware such as Truetime [11], Δ is at least the network's relative diameter in the worst-case [7].

We then show that in the same periods, by *leveraging the asymmetric message delays of geo-distributed networks*, worst-case read blockage times can be below the network's relative diameter at followers close to the leader or the network's center by presenting two new linearizable local read algorithms named *Pairwise-Leader (PL)* and *Pairwise-All (PA)*. With PL, the worst-case read blockage time at a process is the round-trip *relative message delay* between it and the leader, i.e., twice the message delay minus the known lower bound between it and the leader. With PA, the worst-case read blockage time at a process is its *relative eccentricity*, i.e., the maximum message delay minus the known lower bound on message delay between it and all other processes. Both PL and PA enjoy

[1]These assumptions approximate the usual behavior of networks in practice because the variance of message delays is low [18], and failures are infrequent [13].
[2]This precludes the use of specialized clock synchronization hardware.

Proceedings of the VLDB Endowment, Vol. 18, No. 8 ISSN 2150-8097.
doi:10.14778/3742728.3742738

reduced worst-case read blockage times at "well-located" followers because they *do not* rely on globally synchronized clocks to schedule events. Instead, they use a novel event scheduling primitive to schedule events across *pairs of processes* (hence the name *Pairwise*).

To illustrate the above, consider the three-region AWS [1] network shown in Figure 1 where ℓ is the leader. When using the speed of light propagation times as message delay lower bounds, this network's relative diameter is 27 ms. Therefore, with all existing algorithms, the worst-case read blockage time at process p and process q is at least 27 ms. In contrast, the worst-case read blockage time with PL is 10 ms at p and 54 ms at q, because these are the round-trip relative message delays from p and q to ℓ, respectively. PL reduced p's worst-case read blockage time compared to existing algorithms because it is close to ℓ at the cost of increasing q's worst-case read blockage time. While this presents a trade-off, in the extreme scenario where a process r is in the same datacenter as the leader, r's worst-case read blockage time is virtually *zero* with PL. This is because message delays inside the same datacenter are in the order of microseconds [19]. PA, on the other hand, does not present the above trade-off. The worst-case read blockage time with PA is 20 ms at p and 27 ms at q because these are the relative eccentricities[3] of p and q, respectively. Compared to PL, PA reduces worst-case read blockage times at processes far from the leader at the cost of increasing them at processes close to the leader.

Besides our analysis, we compare the performance of our Pairwise algorithms to existing ones by evaluating them on a geo-distributed AWS network. Our experiments show that PL and PA reduce worst-case read latency by up to 50x and 2x, respectively, and increase read throughput by up to 19.4x and 3.3x, respectively. We also implement a distributed version of RocksDB [2] and show that in a YCSB [10] workload, PL and PA increase throughput by 3.6x and 1.2x compared to existing algorithms, respectively.

In summary, this paper makes the following contributions:

- Shows that the worst-case read blockage time with existing algorithms is at least the network's relative diameter.
- Presents two new algorithms that provide worst-case read blockage times below the network's relative diameter at processes close to the leader or the network's center.
- Describes a primitive for scheduling events across pairs of processes without requiring globally synchronized clocks.
- Shows experimentally that compared to existing algorithms, PL (resp. PA) reduces worst-case read latency by up to 50x (resp. 2x), increases read throughput by up to 19.4x (resp. 3.3x), and in a distributed version of RocksDB, increases throughput by 3.6x (resp. 1.2x) in a YCSB workload.

2 BACKGROUND

We begin by describing existing linearizable local read algorithms.

2.1 Leader-Based State Machine Replication

The basis of all linearizable local read algorithms is a state machine replication (SMR) algorithm such as Paxos [21] or Raft [27]. These algorithms provide fault-tolerant replication of a state machine across a set of processes. To do so, each process maintains a copy of the state machine referred to as its *replica*. Replicas are assumed to transition atomically between states, so if two replicas apply the same sequence of operations, they will reach the same state [28]. In this context, operations are defined as deterministic computations that transition the state machine from one state to another and then return any outputs based on these states.

SMR algorithms provide *linearizability* [17]: a consistency model that ensures that operations take effect in some total order consistent with the real-time ordering of operations, i.e., if operation o_1 completes in real-time before operation o_2 begins, then o_1 must take effect before o_2. SMR algorithms typically provide linearizability by *assigning* each operation to a unique natural number known as its *index number*. This assignment is consistent with the operation's real-time ordering, e.g., o_1's index number is smaller than o_2's.

In leader-based SMR algorithms, a process p performs an operation o by forwarding it to the leader, who ensures that all non-faulty processes apply o. It does so in two phases: prepare and commit. The prepare phase starts with the leader assigning o to the next largest index number i and proposing this assignment to all processes. The prepare phase is then complete once a majority of processes acknowledge this proposal. After this, the leader instructs each process to commit o, i.e., apply o to their replica after all operations assigned to index numbers smaller than i have been applied. Finally, p returns the output from applying o to its replica.

2.2 Local Read Algorithms

Generally speaking, local read algorithms build on leader-based SMR algorithms in the following way. They first add a procedure that does not send any messages for performing reads, i.e., operations that *do not* transition the state machine's state. They then slightly modify the two-phase procedure described in §2.1 for performing read-modify-writes (RMW), i.e., operations that *do* transition the state machine's state (all operations other than reads).

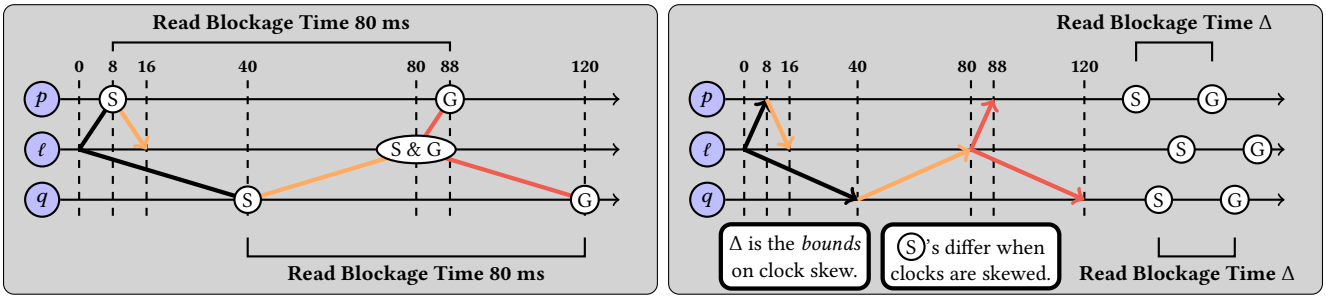
A process p performs a read r by first assigning it to the index number of some RMW. This is so that operations take effect in the following order: the RMW assigned to index 1, the (possibly empty) sequence of reads assigned to index 1, the RMW assigned to index 2, and so forth. After p assigns r to some index i, it returns the output from applying r against its replica once it is at the ith version, i.e., the maximum index assigned to any RMW applied against it is i.

The challenge in performing reads without sending any messages is determining what index number to assign a read to using *only local information*. Local read algorithms overcome this by scheduling a *stop* and *go* event for every index i on every process. A stop event for i on process p represents when p stops assigning reads to indices less than i, whereas a go event for i on p represents when p may apply reads against its replica assigned to i. This generic mechanism ultimately guarantees linearizability by ensuring that all go events for i occur at or after all stop events for i.

To ensure that all go events for i occur at or after all stop events for i in the absence of failures[4], the leader waits to receive acknowledgments from *all* processes, instead of from some *majority* of processes, before sending commit messages for i. This is roughly because, as we will see next, every process p's stop and go event for i occurs when p receives a prepare and commit message for i, respectively, or is delayed by a constant amount from these times.

[3]Note that a process' relative eccentricity is at most the network's relative diameter.

[4]We outline how to tolerate crash failures in §7.



(a) Eager Stamping.

(b) Delayed Stamping.

Figure 2: Eager and Delayed Stamping algorithm's worst-case read blockage time in the network shown in Figure 1. Black, orange, and red arrows represent prepare, acknowledgment, and commit messages, respectively. Numbers above dashed lines are times (in ms) since the leader sent the first prepare (time 0). Stop and go events are shown as (S) and (G), respectively. (S & G) illustrates a combined stop and go event. Subsequent figures are illustrated in the same format, so we refrain from repeating it.

2.3 Scheduling Stop and Go Events

We now describe the three approaches existing local read algorithms use to schedule stop and go events.

In the *Invalidation* approach used by Megastore [5], PQL [26], and Hermes [20], followers apply reads against their replica only when it is *valid*. A follower's replica becomes *invalid* once it receives a prepare message for some index i (this is i's stop event). It then again becomes valid once it receives a commit message for i if no other prepare messages have arrived in the meantime (this is i's go event). The leader is an exception to this because it always knows the state's latest version. Specifically, at any time, the leader performs a read against the version of the state it last sent a commit message for (this is i's stop and go event at the leader).

In the *Eager Stamping* approach used by CHT [9], if a follower begins performing a read and the maximum index it has received a prepare message for is i (this is i's stop event), then the read is *stamped* with i. The read is then applied once the follower has received a commit message for all indices up to and including i (this is i's go event). The leader is an exception to this as it can perform reads in the same way as in the Invalidation approach. We illustrate these events in Figure 2a in the network shown in Figure 1.

Lastly, the *Delayed Stamping* approach used by CockroachDB Global Tables [29] and BHT [6] builds on the Eager Stamping approach by decoupling the occurrence of stop and go events from the reception of prepare and commit messages. They do so by using globally synchronized clocks to determine when RMWs become *visible*. When the leader receives a RMW o, it queries its current clock time t and assigns o a visibility time $v = t + \alpha$ where α is a configuration parameter known as the *visibility delay*. When a process receives a prepare message for index i and visibility time v, it stops stamping reads with indices less than i once v has elapsed on its clock (this is i's stop event). In other words, a process stamps a read with the maximum index whose visibility time has elapsed on its clock at that time. To determine when v has elapsed on every process' clock, they are assumed to *always* be within Δ of each other where Δ is known "from the get-go". A process waits to apply o until it receives commit messages for all indices up to and including i and $v + \Delta$ has elapsed on its clock (this is i's go event). Note that, unlike the previous two approaches, the leader does not have special stop and go events. We illustrate this in Figure 2b.

3 ANALYSIS OF EXISTING ALGORITHMS

We now show that the worst-case read blockage time at every follower with all existing local read algorithms is at least the network's relative diameter when deployed on commodity hardware.

3.1 Assumptions and Notation

Our analysis considers periods where the delay of every message between every two processes p and q is exactly δ_{pq}, and all processes are non-faulty. We also assume that between p and q there is a known lower bound on message delay denoted by $\delta^{\min}_{pq}$, and that all local computation is instantaneous. Moreover, the above quantities are symmetric, i.e., for all p and q, $\delta_{pq} = \delta_{qp}$ and $\delta^{\min}_{pq} = \delta^{\min}_{qp}$.

We use the terms defined in Table 1 to describe the results of our analysis. We define *p's eccentricity* as $E_p = \max_{q \in \Pi} \delta_{pq}$ (where Π is the set of all processes) and the *network's diameter* as $D = \max_{p \in \Pi} E_p$. We also define the *relative message delay* between p and q as $\overline{\delta}_{pq} = \delta_{pq} - \delta^{\min}_{pq}$, *$p$'s relative eccentricity* as $\overline{E}_p = \max_{q \in \Pi} \overline{\delta}_{pq}$, and the *network's relative diameter* as $\overline{D} = \max_{p \in \Pi} \overline{E}_p$. Note that: $\overline{\delta}_{pq} \leq \delta_{pq}$, $\overline{E}_p \leq E_p$, and $\overline{D} \leq D$.

3.2 Analysis

We give the worst-case read blockage times provided by all existing algorithms and our Pairwise algorithms in Table 2. We now justify the values listed for all existing algorithms. The following two sections justify the values listed for our Pairwise algorithms.

Recall that in Invalidation algorithms, follower p performs reads only when its replica is valid, and p's replica is valid if it has received a commit message for every index it has received a prepare message for. So, when p receives prepare messages for multiple indices before receiving a commit message for any of them, p's state is invalid until it receives commit messages for all of them. Consequently, when the leader sends prepare messages quicker than it sends commit messages (because RMWs arrive at the leader more frequently than acknowledgments from all processes do), p's state is perpetually invalid. Therefore, the worst-case read blockage time at p is ∞.

In Eager Stamping algorithms, recall that p stamps a read with the maximum index i it has received a prepare message for, and p applies this read once it has received commit messages for all indices up to and including i. Since a read at p can be stamped with index i

Table 1: Terms used in the paper.

Term	Definition
δ_{pq}	Assumed message delay between p and q
$E_p = \max_{q \in \Pi} \delta_{pq}$	p's eccentricity
$D = \max_{p \in \Pi} E_p$	Network's diameter
$\delta^{\min}_{pq}$	Lower bound on message delay between p and q
$\overline{\delta}_{pq} = \delta_{pq} - \delta^{\min}_{pq}$	Relative message delay between p and q
$\overline{E}_p = \max_{q \in \Pi} \overline{\delta}_{pq}$	p's relative eccentricity
$\overline{D} = \max_{p \in \Pi} \overline{E}_p$	Network's relative diameter

the moment p receives a prepare message for i, reads at p block in the worst-case from this time until p receives a commit message for all indices up to and including i. Furthermore, since message delays are fixed, p receives commit messages in index order. Thus, reads at p block in the worst-case from the time p receives a prepare message for i until it receives a commit message for i. Since the leader sends commit messages for i only after it receives acknowledgments from all processes, the leader sends commit messages for i the maximum round-trip message delay between it and all processes after it sends prepare messages for i. Hence, since prepare and commit messages from the leader to p take the same time, p receives a commit message for i $2 \cdot E_\ell$ after it receives a prepare message for i. Therefore, the worst-case read blockage time at p is $2 \cdot E_\ell$.

Recall that in Delayed Stamping algorithms, p stamps a read with the maximum index i whose visibility time v has elapsed on p's clock, and p applies this read once it has received commit messages for all indices up to and including i and $v+\Delta$ has elapsed on p's clock. Since a read at p can be stamped with index i the moment v elapses on p's clock, reads at p block in the worst-case from this time until p receives a commit message for all indices up to and including i and $v+\Delta$ has elapsed on p's clock. Assuming that p receives commit messages for all indices up to and including i before $v + \Delta$ elapses on its clock, the worst-case read blockage time at p is Δ.

We now show that $\Delta \geq \overline{D}$ when deployed on commodity hardware. Suppose there is an upper-bound on message delay between processes p and q ($\delta^{\max}_{pq}$). Assuming that clocks don't drift, Δ in the worst-case is at least half the maximum *message delay uncertainty*: $\Delta \geq \frac{1}{2} \cdot \max_{p,q \in \Pi} \delta^{\max}_{pq} - \delta^{\min}_{pq}$ [7]. To avoid assuming an incorrect upper bound and violating linearizability, $\delta^{\max}_{pq} - \delta^{\min}_{pq}$ is replaced by the *measured message delay uncertainty* between p and q, i.e., the measured round-trip message delay between p and q minus $2 \cdot \delta^{\min}_{pq}$ [23]. Since in the periods our analysis considers the round-trip message delay between p and q is $2 \cdot \delta_{pq}$, the measured message delay uncertainty is $2 \cdot \overline{\delta}_{pq}$. Therefore $\Delta \geq \overline{D}$.

Finally, we show that the worst-case read blockage time at every follower with all existing local read algorithms is at least the network's relative diameter ($\overline{D}$). This is immediate for Invalidation and Delayed Stamping algorithms. For Eager Stamping algorithms, this is true under the additional assumption that δ respects the triangle inequality (for all processes p, q, and r, $\delta_{pr} \leq \delta_{pq} + \delta_{qr}$). This is because, in Eager Stamping algorithms, a follower's worst-case read blockage time is at least twice the network's radius ($R = \min_{p \in \Pi} E_p$) because by definition $2 \cdot E_\ell \geq 2 \cdot R$. Since δ is assumed to respect the triangle inequality, the network's diameter is at most twice the network's radius ($2 \cdot R \geq D$). Thus, a follower's worst-case read blockage time is at least the network's diameter, which by definition is at least the network's relative diameter ($D \geq \overline{D}$).

Table 2: Read blockage time at process p. ℓ is the leader.

Algorithm	Worst-case read blockage time at $p \neq \ell$
Invalidation ([5, 20, 26])	∞
Eager Stamping ([9])	$2 \cdot E_\ell$
Delayed Stamping ([6, 29])	$\overline{D}$
Pairwise-Leader (§4)	$2 \cdot \overline{\delta}_{p\ell}$
Pairwise-All (§5)	$\overline{E}_p$

4 PAIRWISE-LEADER

We now describe Pairwise-Leader (PL), which achieves worst-case read blockage times below $\overline{D}$ at followers close to the leader ℓ. Specifically with PL, every process p's worst-case read blockage time is the round-trip relative message delay between it and ℓ ($2 \cdot \overline{\delta}_{p\ell}$).

4.1 Overview

To understand how PL achieves this, it is helpful to first consider a stepping-stone algorithm that achieves slightly higher worst-case read blockage times ($2 \cdot \delta_{p\ell}$ vs. $2 \cdot \overline{\delta}_{p\ell}$). The core idea of PL's stepping-stone is to *selectively delay when prepare messages are sent in Eager Stamping algorithms.* Our goal in doing this is to decrease the time between when a process p receives a prepare message for each index i and when p receives a commit message for i. This ultimately reduces p's worst-case read blockage time because, as our analysis of Eager Stamping algorithms showed, a read at p stamped with i blocks in the worst-case from the time p receives a prepare message for i until it receives a commit message for i.

To achieve this goal, prepare messages are delayed so that the leader receives acknowledgments from all processes at the same time (under the assumptions outlined in §3.1). Specifically, a process p's prepare message is delayed by twice the leader's eccentricity minus the round-trip message delay between p and the leader ℓ. For example, in the network shown in Figure 1, $2 \cdot E_\ell$ is 80 ms and $2 \cdot \delta_{p\ell}$ is 16 ms so p's prepare message is delayed by 64 ms.

As Figure 3 shows, by delaying the sending of p's prepare message, p's prepare and commit messages differ by 16 ms. This is because when the leader receives acknowledgments from all processes at the same time T, every process p's prepare message occurs the message delay between it and the leader before T, and p's commit message occurs the message delay between it and the leader after T. Hence, p's prepare and commit messages differ in real-time by $2 \cdot \delta_{p\ell}$ and so, p's worst-case read blockage time is $2 \cdot \delta_{p\ell}$.

Since our goal with PL is $2 \cdot \overline{\delta}_{p\ell}$, what remains is to shave off an additional $2 \cdot \delta^{\min}_{p\ell}$ from PL's stepping stone. Recall that in §2, we saw one approach for reducing worst-case read blockage times by the message delay lower bounds: *decouple the occurrence of stop and go events from the reception of messages.* This enabled Delayed Stamping algorithms to reduce the worst-case read blockage times provided by Eager Stamping algorithms from twice the leader's eccentricity ($2 \cdot E_\ell$) to the network's relative diameter ($\overline{D}$). By roughly applying this idea to PL's stepping stone, we obtain PL.

The challenge in applying this idea is ensuring that stop and go events "maintain their shape" after decoupling. To see why, recall that the Delayed Stamping algorithms assign a visibility time v to each index i, such that every process' stop event for i occurs at v and its go event for i occurs at least at $v + \Delta$. Since $\Delta \geq \overline{D}$, decoupling stop and go events in this way results in their difference

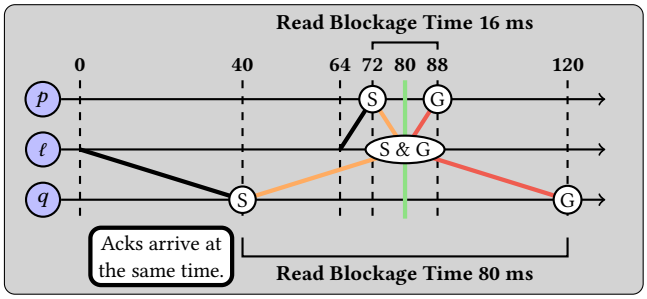


Figure 3: A stepping stone to Pairwise-Leader.

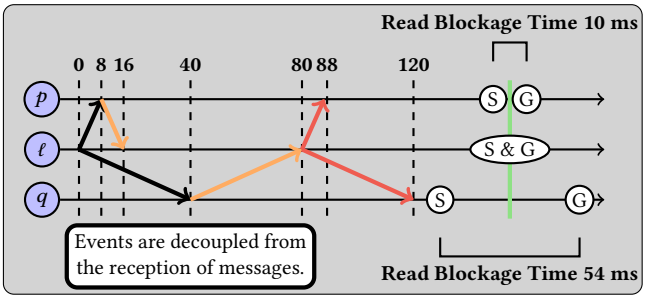
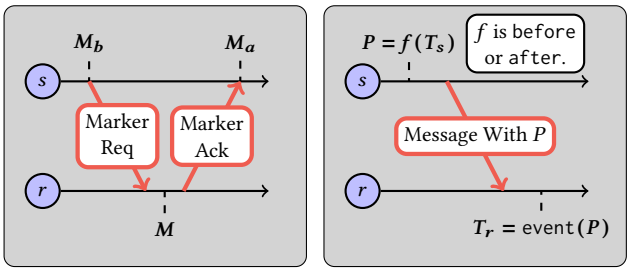


Figure 4: Pairwise-Leader.



(a) Establishing markers. **(b) Scheduling an event.**

Figure 5: The two event scheduling primitive interactions.

being at least $\overline{D}$ on every process. Therefore, to achieve our goal, *we need a new approach for decoupling stop and go events.*

PL decouples stop and go events using a novel event scheduling primitive. This primitive enables events to be scheduled across a pair of processes p and q that are *guaranteed* to occur before or after each other in real-time as desired. The key feature of this primitive is that under the assumptions outlined in §3.1, events between p and q differ in real-time by $\overline{\delta}_{pq}$. The leader in PL uses this primitive to schedule every process p's stop event for index i *before* i's visibility time and p's go event for i *after* i's visibility time. We illustrate this in Figure 4 where the green line represents i's visibility time. As we will show, scheduling events this way results in every process p's worst-case read blockage time being $2 \cdot \overline{\delta}_{p\ell}$.

4.2 The Event Scheduling Primitive

We begin by explaining how this primitive provides a process s, known as the *sender*, the ability to compute a time T_r on a process r's clock, known as the *receiver*, such that T_r occurs before or after T_s on the sender's clock in real-time as desired. Note that when we say T_r occurs before or after T_s, they could occur at the same real-time.

At the high level, the primitive provides this functionality by establishing causally ordered timestamps called *markers*. The sender has a before marker M_b and an after marker M_a, while the receiver has a single marker M, such that M_b occurs before M, and M occurs before M_a in real-time. To schedule T_r, the sender computes the elapsed time D between T_s and one of its markers, followed by relaying D to the receiver. The receiver then computes T_r as $D + M$.

To establish markers, the sender first records its clock time as its before marker M_b. The sender then sends a *marker request* to the receiver. Once received, the receiver records its clock time as its marker M. The receiver then sends the sender a *marker acknowledgment*. Once received, the sender records its clock time as its after marker M_a. At this point, the set of markers is established, and by causality, M_b occurs before M in real-time, and M occurs before M_a in real-time. We illustrate this interaction in Figure 5a.

After a set of markers is established, the sender can schedule events. It does so by computing D based on whether T_r should occur before or after T_s. We first describe computing D assuming that clocks do not drift and then with bounded drift afterward.

If T_r should occur before T_s in real-time, then $D = T_s - M_a + \delta^{\min}_{rs}$. To see why this guarantees that T_r occurs before T_s, observe that when clocks don't drift, the real-time difference between T_s on the sender's clock and $T_s - M_a + M$ on the receiver's clock, is the same as the real-time difference between M_a and M. Since M occurs at least $\delta^{\min}_{rs}$ before M_a in real-time, $T_s - M_a + M$ on the receiver's clock occurs at least $\delta^{\min}_{rs}$ before T_s on the sender's clock in real-time. Hence, T_r ($T_s - M_a + \delta^{\min}_{rs} + M$) occurs before T_s in real-time. In the other case where T_r should occur after T_s in real-time, then $D = T_s - M_b - \delta^{\min}_{sr}$ for similar reasons.

We now describe computing D assuming ϵ-bounded clock drift.[5] Let $C_p(t)$ denote process p's hardware clock time at real-time t; p's clock has ϵ-bounded drift for some $\epsilon \in [0, 1]$ if for all real-times $t_1 \le t_2$, $(1-\epsilon)(t_2 - t_1) \le C_p(t_2) - C_p(t_1) \le (1+\epsilon)(t_2 - t_1)$. To account for this, the above computations for D are scaled proportionally to ϵ. Specifically, if T_r should occur before T_s in real-time, then $D = (1-\epsilon)(\frac{T_s - M_a}{1+\epsilon} + \delta^{\min}_{rs})$. This is because the receiver's clock can tick as slowly as $(1-\epsilon)$ while the sender's clock can tick as fast as $(1+\epsilon)$. To account for this, $T_s - M_a$ is scaled by $\frac{1-\epsilon}{1+\epsilon}$. Conversely, the lower bound is scaled by $1-\epsilon$ since $\delta^{\min}_{rs}$ is perceived only by the receiver. In the other case where T_r should occur after T_s in real-time, $D = (1+\epsilon)(\frac{T_s - M_b}{1-\epsilon} - \delta^{\min}_{sr})$ for similar reasons.

We now show that under the assumptions outlined in §3.1, if $\epsilon = 0$ then T_r and T_s differ in real-time by $\overline{\delta}_{sr}$. If T_r should occur before T_s, then as discussed above, since clocks do not drift, the real-time difference between T_s on the sender's clock and $T_s - M_a + M$ on the receiver's clock is the same as the real-time difference between M_a and M. Hence, the real-time difference between T_s and $T_s - M_a + M$ is δ_{rs}. Furthermore, since $T_r = T_s - M_a + \delta^{\min}_{rs} + M$, this difference shrinks by a factor of $\delta^{\min}_{rs}$. Therefore, the real-time difference between T_r and T_s is $\overline{\delta}_{rs}$ which by definition is equal to $\overline{\delta}_{sr}$. In the other case where T_r should occur after T_s, the real-time difference between T_r and T_s is $\overline{\delta}_{sr}$ for similar reasons.

In the general case where $\epsilon \ge 0$, the real-time difference between T_r and T_s grows over time. This is because if T_r should occur before T_s, then the real-time difference between T_r and T_s grows as a function of $T_s - M_a$ since it is scaled by $\frac{1-\epsilon}{1+\epsilon}$. Hence, to keep the real-time difference between T_r and T_s as small as possible, $T_s - M_a$

[5] In practice, ϵ is usually defined as 200 parts per million (ppm) [11, 24].

needs to be as small as possible. We achieve this by periodically re-establishing markers. This requires a slight modification to event scheduling as D can now be computed using multiple markers. Each set of markers is assigned a unique version number to distinguish between them. Then, to schedule T_r before or after T_s, the sender invokes the corresponding procedure `before`(T_s) or `after`(T_s). These procedures return a payload P containing D computed using the most recently established set of markers before T_s, along with its version number. The receiver then computes T_r by invoking `event`(P), which adds D to the corresponding version of M. This interaction is illustrated in Figure 5b. We note that by frequently re-establishing markers, the real-time difference between T_r and T_s is virtually $\overline{\delta}_{sr}$. We confirm this experimentally in §8.

4.3 The Algorithm

We now describe PL in detail (Algorithm 1). We do so without failures to highlight our main ideas and avoid rehashing ideas from prior work. We outline how to tolerate crash failures in §7.

Algorithm 1 assumes a *timed asynchronous system* consisting of a finite set of non-faulty processes Π [12]. This is the standard asynchronous system in which messages passed between processes may be arbitrarily delayed but not lost with the addition that processes are equipped with hardware clocks with bounded drift. In the version of the algorithm that tolerates failures, to guarantee liveliness, there must be a known maximum message delay $\delta^{\max}$.

Since Algorithm 1 uses the event scheduling primitive, the user must define ϵ and $\delta^{\min}_{pq}$ for all processes p and q. To guarantee linearizability, every process' clock must have ϵ-bounded drift, and every message between process p and q must take at least $\delta^{\min}_{pq}$. The user must also define a non-negative visibility delay α. As we will see, α must be sufficiently large to achieve our desired worst-case read blockage times. Smaller values will increase worst-case read blockage times, and larger values will increase RMW latency (see §8). Note that the choice of α does not impact correctness.

We now walk through Algorithm 1. We start with a few remarks followed by describing how operations are performed. Each process runs three threads in parallel, and each performs different jobs. Thread 1 performs RMWs (line 1-6) and reads (line 7-12), Thread 2 handles RMW (line 13-20), prepare (line 21-25), and acknowledgment messages (line 26-27), and Thread 3 handles commit messages (line 28-44). Each thread performs one job at a time until completion.

Processes maintain *multi-versioned* replicas by applying operations via the following two procedures. The first is `Apply`(i, o), which creates the ith version of the state and returns any outputs by applying a RMW operation o against the $(i-1)$th version. The second is `Read`(i, o), which applies a read operation o against the ith version of the state once it is created and returns any outputs.

To schedule stop and go events, PL uses multiple instances of the event scheduling primitive. This is because the leader schedules these events on every process, so the leader is a sender, and each process is a receiver. We discern which instance of the event scheduling primitive a process is using through the notation (s, r).`proc`, which means `proc` is being invoked on the instance of the event scheduling primitive where s is the sender and r is the receiver.

We now describe how a process p performs operations. To perform a RMW o, p invokes RMW(o) (line 1). This procedure begins

```
   Thread 1:
1  procedure RMW(o) at p:
2      c := ++Cntr                    /* local operation counter, initially 0 */
3      op := (o, (p, c))                                  /* unique operation */
4      send ⟨RMW, op⟩ to ℓ                              /* send op to leader */
5      wait until resp(op) ≠ NULL  /* op's response, initially NULL */
6      return resp(op)              /* return response written on line 43 */
7  procedure read(o):
8      t := now()                              /* current hardware clock time */
9      (lb, ub) := (CIndex, PIndex)      /* range of stop events to check */
10     j := max{i | lb ≤ i ≤ ub ∧ Stop[i] ≤ t}          /* version to read */
11     wait until CIndex ≥ j        /* wait for jth version to be created */
12     return Read(j, o)                        /* apply o against jth version */
   Thread 2:
13 upon receiving ⟨RMW, op⟩:
14     t := now()                              /* current hardware clock time */
15     i := ++Index                              /* latest index, initially 0 */
16     Operation[i] := op                  /* record operation assigned to i */
17     foreach p ∈ Π do                    /* Π is the set of all processes */
18         P_b := (ℓ, p).before(t + α)                /* p's stop event for i */
19         P_a := (ℓ, p).after(t + α)                 /* p's go event for i */
20         send ⟨PREPARE, i, P_b, P_a⟩ to p           /* send prepare to p */
21 upon receiving ⟨PREPARE, i, P_b, P_a⟩ at q:
22     Stop[i] := (ℓ, q).event(P_b)       /* stop event for i, initially ∞ */
23     Go[i] := (ℓ, q).event(P_a)                        /* go event for i */
24     PIndex := max(PIndex, i) /* maximum prepared index, initially 0 */
25     send ⟨ACK, i⟩ to ℓ                                /* send ack to ℓ */
26 upon receiving ⟨ACK, i⟩ from Π:
27     send ⟨COMMIT, i, Operation[i]⟩ to Π
   Thread 3:
28 upon receiving ⟨COMMIT, i, op⟩:
29     commit_or_queue(i, op)
30 procedure commit_or_queue(i, op):
31     if CIndex ≠ i − 1 then      /* latest committed index, initially 0 */
32         CommitQueue.insert((i, op))     /* priority queue by index */
33     else                                              /* CIndex = i − 1 */
34         commit(i, op)                                    /* commit op */
35         expected := i + 1                    /* next index to commit */
36         while CommitQueue.min() = (expected, −) do
37             (i′, op′) := CommitQueue.extract_min()
38             commit(i′, op′)                              /* commit op′ */
39             expected++           /* look for next contiguous index */
40 procedure commit(i, op):
41     wait until now() ≥ Go[i]                /* go event occurs for i */
42     (o, (−, −)) := op                          /* extract o from op */
43     resp(op) := Apply(i, o)          /* apply o to the i − 1th version */
44     CIndex = i                       /* advance latest committed index */
```

Algorithm 1: PL without failures. ℓ is the leader.

by creating a globally unique operation op corresponding to this *instance* of o, i.e., p's cth time invoking the RMW procedure (line 2-3). Afterward, p sends a RMW message for op to the leader ℓ (line 4).

Upon receiving this message, the leader queries its current clock time t and assigns op to the next index number i (line 14-16). The leader then schedules a stop and go event for i on every process to occur before and after i's visibility time $t + \alpha$ using the event scheduling primitive's `before`$(t + \alpha)$ and `after`$(t + \alpha)$ procedures (line 17-19). The leader then sends prepare messages for i (line 20).

When a process q receives this message, it computes its stop and go event for i using the `event` procedure provided by the event scheduling primitive (line 22-23). q then updates $PIndex$, the maximum prepared index so far, and sends an acknowledgment to the leader (line 24-25). Once the leader has received acknowledgments from all processes, it sends commit messages for i (line 27).

When q receives this message, it checks the latest committed index $CIndex$ (line 31). If it is not $i - 1$, then (i, op) is added to the *commit queue*: a priority queue ordered by index number (line 32). Otherwise, op is committed, followed by committing all index contiguous operations in the commit queue (line 35-39). To commit op,

q first waits for its go event for i to occur (line 41). q then extracts the RMW o from op, applies o to its replica, stores its response in $resp(op)$, and updates its latest committed index (line 42-44). Finally, once p commits op, it returns $resp(op)$ (line 5-6).

To perform a read o, p invokes read(o) (line 7). This procedure queries p's current clock time t (line 8) and copies the current values of $CIndex$ and $PIndex$ as lb and ub, respectively (line 9). o is then stamped with the maximum index in $[lb, ub]$ whose stop event has occurred as of t (line 10); let j be this index. Finally, once the RMW assigned to j has been applied (line 11), o is applied against the jth version of the state, and its output is returned (line 12).

4.4 Analysis

We now show that under the assumptions outlined in §3.1 if $\epsilon = 0$ and $\alpha \geq 2 \cdot E_\ell + E_\ell^{\min}$ where $E_\ell^{\min} = \max_{q \in \Pi} \delta_{\ell q}^{\min}$, then every process p's worst-case read blockage time is $2 \cdot \overline{\delta}_{p\ell}$. Recall that our analysis assumes that all local computation is instantaneous. In Algorithm 1, this means that only two steps take any real-time. The first is when a process p sends a message to a process q, q's **upon receiving** step happens δ_{pq} real-time after p's **send** step. The second is how long a process spends on a **wait until** step.

Consider some process p's invocation of read(o) stamped with j on line 10. We start with two observations about any $i \in [1, j]$ that follow from our assumptions that $\epsilon = 0$ and messages are not lost, respectively. Let op_i be the RMW assigned to index i on line 15. The first is that op_i's visibility time V_i occurs in real-time at $T_i + \alpha$ where T_i is the real-time the leader received op_i on line 13. The second is that p wrote timestamps into $Stop[i]$ and $Go[i]$ on lines 22 and 23 which occur at real-times S_i and G_i, respectively.

We now relate these times. Since p's stop event for i is scheduled to occur before op_i's visibility time on line 18, and $\epsilon = 0$, by our analysis in §4.2 $S_i = V_i - \overline{\delta}_{p\ell}$. Likewise, since p's go event for i is scheduled to occur after op_i's visibility time on line 19, $G_i = V_i + \overline{\delta}_{p\ell}$.

The rest of our analysis is done in two steps. The first is to show that p applies op_j on line 43 at G_j. This follows from two facts: (1) for every $i \in [1, j)$ $G_i \leq G_{i+1}$ and (2) for every $i \in [1, j]$ p receives a commit message for i by G_i. To see why, notice that (1) and (2) imply that p receives a commit message for all $i \in [1, j]$ by G_j. Thus, by the commit_or_queue procedure, p invokes commit(j, op_j) by G_j. Therefore, line 41 finishes at G_j so p applies op_j on line 43 at G_j.

We now justify (1) and (2). For (1), first notice that the leader receives RMWs on line 13 in index order and so, $T_i \leq T_{i+1}$. Since $V_i = T_i + \alpha$ this implies that $V_i \leq V_{i+1}$ and thus, $V_i + \overline{\delta}_{p\ell} \leq V_{i+1} + \overline{\delta}_{p\ell}$. Finally, since $G_i = V_i + \overline{\delta}_{p\ell}$ it follows that $G_i \leq G_{i+1}$.

For (2), first notice that p receives a commit message for i at $T_i + 2 \cdot E_\ell + \delta_{\ell p}$. This is because the leader receives acknowledgments for i from all processes on line 26 at $T_i + 2 \cdot E_\ell$ and p receives a commit message for i on line 28 after an additional $\delta_{\ell p}$. What remains is to show that $G_i \geq T_i + 2 \cdot E_\ell + \delta_{\ell p}$. Recall that (a) $G_i = V_i + \overline{\delta}_{p\ell}$, (b) $V_i = T_i + \alpha$, and (c) $\alpha \geq 2 \cdot E_\ell + E_\ell^{\min}$, and observe that (d) $E_\ell^{\min} + \overline{\delta}_{\ell p} \geq \delta_{\ell p}$. By (a), (b), and (c): $G_i \geq T_i + 2 \cdot E_\ell + E_\ell^{\min} + \overline{\delta}_{p\ell}$ and therefore by (d) $G_i \geq T_i + 2 \cdot E_\ell + \delta_{\ell p}$.

The second step is to show that the real-time of read(o) is at most $G_j - S_j$. First, observe that the real-time of read(o) is how long it waits on line 11. This is because lines 8 to 10 are local computations, and when the wait on line 11 finishes, the jth version of the state has already been created, so line 12 does not wait. Now observe that $CIndex \geq j$ at G_j. This is because $CIndex$ is monotonically increasing, and by step one, $CIndex$ is set to j on line 44 at G_j. These two observations together imply that the real-time of read(o) is the time between when line 11 is invoked and G_j. Since o is stamped with index j on line 10, the earliest line 11 can be invoked is S_j. Therefore, the real-time of read(o) is at most $G_j - S_j$.

Recall that $S_j = V_j - \overline{\delta}_{p\ell}$ and $G_j = V_j + \overline{\delta}_{p\ell}$ so $G_j - S_j = 2 \cdot \overline{\delta}_{p\ell}$. Therefore by step two p's worst-case read blockage time is $2 \cdot \overline{\delta}_{p\ell}$.

5 PAIRWISE-ALL

The last section showed that with PL, every process p's worst-case read blockage time is $2 \cdot \overline{\delta}_{p\ell}$. Therefore, when a process q is in the same datacenter as the leader, q's worst-case read blockage time with PL is virtually zero compared to at least $\overline{D}$ with all existing algorithms. However, when q is far from the leader, q's worst-case read blockage time can be higher than some existing algorithms. For example, if $\overline{\delta}_{q\ell} = \overline{D}$ then with PL q's worst-case read blockage time is $2 \cdot \overline{D}$ compared to just $\overline{D}$ with Delayed Stamping algorithms.

We now describe Pairwise-All (PA), which achieves worst-case read blockage times below $\overline{D}$ at some followers and at most $\overline{D}$ at all followers. Specifically, with PA, every process p's worst-case read blockage time is its relative eccentricity ($\overline{E}_p$). Compared to PL, PA reduces worst-case read blockage times at processes far from the leader, but increases them at processes close to the leader.

5.1 Overview

As in our overview of PL, to understand how PA achieves this, we first consider a stepping-stone algorithm that achieves slightly higher worst-case read blockage times (E_p vs. $\overline{E}_p$). The core idea of PA's stepping stone is the same as PL's: selectively delay when prepare messages are sent in Eager Stamping algorithms. However, in contrast to PL's stepping stone, which delays prepare messages so that the leader receives acknowledgments at the same time, PA's stepping stone delays prepare messages so that they all arrive at the same time (under the assumptions outlined in §3.1). Specifically, a process p's prepare message is delayed by the leader's eccentricity minus the message delay between p and the leader ℓ. For example, in the network shown in Figure 1, E_ℓ is 40 ms and $\delta_{p\ell}$ is 8 ms, so p's prepare message is delayed by 32 ms as shown in Figure 6.

PA's stepping stone also differs from PL's in that *acknowledgments are sent to all processes.* This is instead of sending acknowledgments only to the leader who then sends a commit message to each process after it receives acknowledgments from all processes. Notice that these two approaches are semantically equivalent. This is because they both inform every process when it is safe to commit a RMW o, i.e., when all processes have received a prepare message for o. However, the benefit of sending acknowledgments to all processes is that p can commit o quicker. Specifically, since all processes receive prepare messages for o at the same time, p can commit o its eccentricity after it receives a prepare message for o. Therefore, p's worst-case read blockage time is E_p.

To achieve our goal of $\overline{E}_p$, we apply the same idea to PA's stepping stone as we did with PL's: decouple the occurrence of stop and

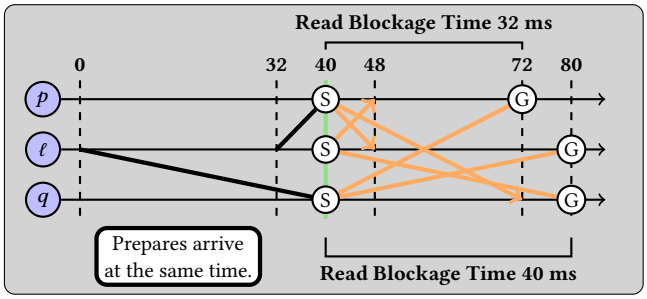


Figure 6: A stepping stone to Pairwise-All.

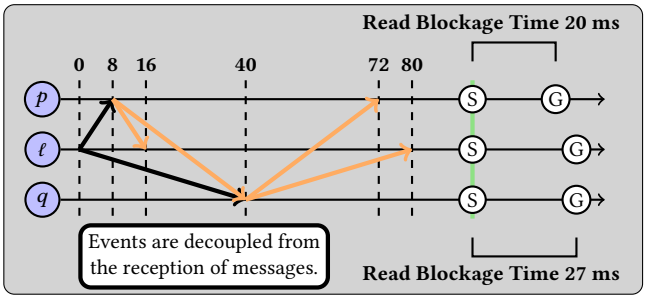


Figure 7: Pairwise-All.

go events from the reception of messages. However, since PA wants all stop events to occur at the same time, we need to enhance the event scheduling primitive with this functionality. The leader in PA then uses this enhanced functionality to schedule every process p's stop event for index i to occur approximately at i's visibility time. Then, every process q uses this primitive to schedule a *stopped* event for i on p that occurs after q's stop event for i. p's go event for i is then the maximum over all stopped events for i on p. We illustrate this in Figure 7 where the green line represents i's visibility time. We emphasize that the leader's accuracy in scheduling p's stop event for i does not impact correctness. This is because p's go event for i is the maximum of all stopped events for i over all processes, which is guaranteed to occur after all stop events for i. Furthermore, as we will show, scheduling events in this way results in every process p's worst-case read blockage time being $\overline{E}_p$.

5.2 Enhancing the Event Scheduling Primitive

We now explain how to provide the sender s with the ability to compute a time T_r on the receiver r's clock such that T_r occurs approximately at T_s on the sender's clock in real-time.

To see how, first recall that the sender has a before marker M_b and an after marker M_a, while the receiver has a single marker M, such that M_b occurs before M in real-time, and M occurs before M_a in real-time. Also, recall that, to schedule T_r, the sender computes the elapsed time D between T_s and its markers, followed by relaying D to the receiver. The receiver then computes T_r as $D + M$.

The sender schedules T_r approximately at T_s by computing D as T_s minus the midpoint between M_b and M_a, i.e., $T_s - \frac{1}{2}(M_b + M_a)$. This is because when message delays are fixed, $\frac{1}{2}(M_b + M_a)$ on the sender's clock occurs at the same real-time as M on the receiver's clock. Thus, if $\epsilon = 0$ then T_r occurs at the same real-time as T_s. Like in §4.2, the primitive exposes this functionality through a procedure called $\texttt{at}(T_s)$ which returns a payload P that contains D computed using the most recently established set of markers before T_s along with its version number. The receiver then computes T_r by invoking $\texttt{event}(P)$ which adds D to the corresponding version of M.

5.3 The Algorithm

We now describe PA in detail (Algorithm 2). PA assumes the same system as PL. Algorithm 2 builds on Algorithm 1 by reusing Thread 1 and the commit_or_queue procedure but differs in how RMWs are performed. We now describe how PA performs RMWs.

To perform a RMW o, a process p invokes RMW(o). This procedure first creates a unique operation op and sends it to the leader. Upon receiving this message, the leader records its current clock

Thread 2:

45 **upon receiving** $\langle RMW, op\rangle$:
46 $\quad t := now()$ /* current time */
47 $\quad i := ++Index$ /* latest index, initially 0 */
48 $\quad$ **foreach** $p \in \Pi$ **do** /* Π is the set of all processes */
49 $\quad\quad P := (\ell, p).\text{at}(t + \alpha)$ /* p's stop event for i */
50 $\quad\quad$ **send** $\langle PREPARE, i, op, P\rangle$ to p /* send prepare to p */
51 **upon receiving** $\langle PREPARE, i, op, P\rangle$ at p:
52 $\quad Operation[i] := op$ /* record operation assigned to i */
53 $\quad Stop[i] := (\ell, p).\text{event}(P)$ /* stop event for i */
54 $\quad PIndex := \max(PIndex, i)$ /* max prepare index, initially 0 */
55 $\quad$ **foreach** $q \in \Pi$ **do**
56 $\quad\quad P' := (p, q).\text{after}(Stop[i])$ /* q's stopped event for i by p */
57 $\quad\quad$ **send** $\langle ACK, i, P'\rangle$ to q /* send ack to q */
58 **upon receiving** $\langle ACK, i, P'\rangle$ at p from q:
59 $\quad Stopped[i][q] := (q, p).\text{event}(P')$ /* stopped event */

Thread 3:

60 **upon receiving** $\langle ACK, i, -\rangle$ from Π:
61 $\quad$ **wait until** $\forall q \in \Pi\ Stopped[i][q] \neq NULL$ /* wait for Thread 2 */
62 $\quad Go[i] := \max_{q \in \Pi} Stopped[i][q]$ /* go event for i */
63 $\quad$ commit_or_queue(i, $Operation[i]$)

Algorithm 2: PA without failures. ℓ is the leader.

time t and assigns op to the next index number i (line 46-47). The leader then schedules a stop event for i on every process to occur approximately at i's visibility time. It does so by invoking the $\texttt{at}(t + \alpha)$ procedure provided by the event scheduling primitive (line 49). The leader then sends prepare messages for i (line 50).

When a process p receives this message it stores the operation and computes its stop event using the event procedure provided by the event scheduling primitive (line 52-53). Next, p updates the maximum index it has received so far (line 54). Then, p schedules a stopped event for i to occur after its stop event for i for each process using the $\texttt{after}(Stop[i])$ procedure (line 56). Lastly, p sends an acknowledgment message for i (line 57).

When a process p receives this message from a process q it computes its stopped event for i from q using the event procedure (line 59). Finally, once p has received acknowledgments for i from all processes, it computes its go event for i as the maximum stopped event received by all processes (line 62) and invokes commit_or_queue as described on line 30 of Algorithm 1.

5.4 Analysis

We now show that under the assumptions outlined in §3.1 if $\epsilon = 0$ and $\alpha \geq E_\ell + D^{\min}$ where $D^{\min} = \max_{p \in \Pi} E_p^{\min}$ then every process p's worst-case read blockage time is $\overline{E}_p$.

Consider some process p's invocation of read(o) stamped with j on line 10. We start with two observations about any $i \in [1, j]$, which follow from our assumptions that $\epsilon = 0$, messages are not lost, and all processes are non-faulty. Let op_i be the RMW assigned

to index i on line 47. The first is that op_i's visibility time V_i occurs in real-time at $T_i + \alpha$ where T_i is the real-time the leader received op_i on line 45. The second is that p wrote timestamps into $Go[i]$ and $Stopped[i][q]$, for every process q, on lines 62 and 59 which occur at real-times $Go(p, i)$ and $Stopped(p, i, q)$, respectively, and every process q wrote a timestamp into $Stop[i]$ on line 53 which occurs at real-time $Stop(q, i)$. We now relate these times.

Since process q's stop event for i is scheduled at op_i's visibility time on line 49, and $\epsilon = 0$, by our analysis in §5.2 $Stop(q, i) = V_i$. Furthermore, since q schedules p's stopped event for i after q's stop event for i on line 56, by our analysis in §4.2 $Stopped(p, i, q) = Stop(q, i) + \overline{\delta}_{qp}$. Finally, since p's go event for i is p's maximum stopped event for i over all processes (line 62), $Go(p, i) = V_i + \overline{E}_p$.

Our analysis is done in the same two steps as in §4.4. The first is to show that p applies op_j on line 43 at $Go(p, j)$. This follows from two facts: (1) for every $i \in [1, j)$ $Go(p, i) \leq Go(p, i + 1)$ and (2) for every $i \in [1, j]$ p receives an acknowledgment message for i from every process by $Go(p, i)$. To see why, notice that (1) and (2) imply that for every $i \in [1, j]$ p invokes commit_or_queue(i, op_i) by $Go(p, j)$, and therefore p applies op_j on line 43 at $Go(p, j)$.

The argument for (1) is similar to that in §4.4. For (2), first notice that p receives acknowledgment messages for i from every process at $T_i + \max_{q \in \Pi} \delta_{\ell q} + \delta_{qp}$. This is because every process q receives a prepare message for i on line 51 at $T_i + \delta_{\ell q}$ and p receives an acknowledgment message for i from q on line 58 after an additional δ_{qp}. What remains is to show that $Go(p, i) \geq T_i + \max_{q \in \Pi} \delta_{\ell q} + \delta_{qp}$. Recall that (a) $Go(p, i) = V_i + \overline{E}_p$, (b) $V_i = T_i + \alpha$, and (c) $\alpha \geq E_\ell + D^{\min}$, and observe that (d) $D^{\min} + \overline{E}_p \geq E_p$. By (a), (b), and (c): $Go(p, i) \geq T_i + E_\ell + D^{\min} + \overline{E}_p$ and therefore by (d) $Go(p, i) \geq T_i + E_\ell + E_p = T_i + \max_{q \in \Pi} \delta_{\ell q} + \delta_{qp}$.

The second step is to show that the real-time of read(o) is at most $Go(p, j) - Stop(p, j)$. Again, the argument is similar to that in §4.4 so we omit it for brevity. Now recall that $Stop(p, j) = V_j$ and $Go(p, j) = V_j + \overline{E}_p$ so $Go(p, j) - Stop(p, j) = \overline{E}_p$. Therefore, by step two p's worst-case read blockage time is $\overline{E}_p$.

6 CORRECTNESS SKETCH

We now sketch why Algorithms 1 and 2 are linearizable. Recall that this requires all operations to take effect in some total order that is consistent with the real-time ordering of operations, i.e., if operation o_1 completes before operation o_2 begins, then o_1 must take effect before o_2. Operations in PL and PA take effect in the same total order: the RMW assigned to index 1, the (possibly empty) sequence of reads stamped with index 1, the RMW assigned to index 2, and so forth. This is ultimately enforced by the `Apply` and `Read` procedures. We now show that this ordering is consistent with the real-time ordering of operations. Consider any operations o_1 and o_2 such that o_1 completes before o_2 begins. There are four cases.

Case 1: o_1 and o_2 are RMWs. Since o_1 is complete, the process p that invoked o_1 executed `Apply`(i_1, o_1) on line 43 where i_1 is o_1's index. First notice that $Index = i_1$ on the leader before p executed `Apply`(i_1, o_1). This is because p executes `Apply`(i_1, o_1) after receiving a commit message for i_1 on line 28 in PL or receiving acknowledgments for i_1 from all processes on line 60 in PA. Hence, the leader sent prepare messages for i_1 beforehand and therefore set $Index = i_1$ before p executed `Apply`(i_1, o_1). Now, since o_1 completed before o_2 begins in real-time, this implies that $Index = i_1$ on the leader before o_2 begins. Since $Index$ is monotonically increasing, this implies that the leader assigns o_2 to an index larger than i_1. Therefore, o_1 is before o_2 in the above total order.

Case 2: o_1 is a RMW and o_2 is a read. Let processes p_1 and p_2 be the processes that invoked o_1 and o_2, respectively. Since o_1 is complete, p_1 executed `Apply`(i_1, o_1) on line 43 at real-time A where i_1 is o_1's index. Hence, line 41 finished at some real-time $Go(p_1, i_1) \leq A$ where $C_{p_1}(Go(p_1, i_1)) \geq Go[i_1]$. We now show that p_2 wrote a timestamp into $Stop[i_1]$ and $PIndex \geq i_1$ before or at $Go(p_1, i_1)$. Since line 41 finishes on p_1 at $Go(p_1, i_1)$, commit(i_1, op_1) was executed on line 40 where op_1 is the unique operation created on line 3 for o_1. In PL, this implies that p_1 received a commit message for i_1 on line 28, and so the leader received acknowledgments for i_1 from all processes on line 26. In PA, this implies that p_1 received acknowledgments for i_1 from all processes on line 58. Hence, p_2 executed $Stop[i_1] = C_{p_2}(Stop(p_2, i_1))$ and $PIndex \geq i_1$ before or at $Go(p_1, i_1)$. Now, observe that in both PL and PA, $Go(p_1, i_1) \geq Stop(p_2, i_1)$. This follows from the real-time ordering guarantees provided by the event scheduling primitive and the definitions of $Go[i_1]$ on p_1 and $Stop[i_1]$ on p_2. Now consider p_2's invocation of read(o_2) stamped with i_2 on line 10. In read(o_2), p_2 first queries its hardware clock on line 8 which returns a timestamp $t = C_{p_2}(R)$. Since p_2's invocation of read(o_2) occurs after o_1 completes, $R \geq A$, and since $A \geq Go(p_1, i_1)$, $R \geq Go(p_1, i_1)$. p_2 then computes lb and ub on line 9. Since $PIndex \geq i_1$ before or at $Go(p_1, i_1)$, $PIndex$ is monotonically increasing, and ub is computed after $Go(p_1, i_1)$, $ub \geq i_1$. There are now two cases to consider. If $lb \geq i_1$ then by line 10 $i_2 \geq lb$ so $i_2 \geq i_1$. Otherwise, $i_1 > lb$. Since $R \geq Go(p_1, i_1)$ and $Go(p_1, i_1) \geq Stop(p_2, i_1)$, $R \geq Stop(p_2, i_1)$. Hence, since C_{p_2} is monotonically increasing, $t \geq Stop[i_1]$ and so by line 10 $i_2 \geq i_1$. Therefore, o_1 is before o_2 in the above total order.

For brevity, we omit arguments for case 3 (o_1 is a read operation and o_2 is a RMW operation) and case 4 (o_1 and o_2 are read operations) because you can satisfy both by first identifying the RMW operation that was assigned to the same index that o_1 was stamped with, then use the above argument for case 1 to satisfy case 3 and use the above argument for case 2 to satisfy case 4.

7 FAULT TOLERANCE

We now outline how to extend our failure-free descriptions of existing algorithms (§2) and the Pairwise algorithms (§4 and §5) to tolerate $f < \frac{|\Pi|}{2}$ crash failures using ideas from prior work [6, 9].

We start with how to tolerate follower failures. As we have seen, in the absence of failures, all followers must acknowledge a prepare message for index i before any process can commit the RMW o assigned to i. This is so that all followers stop stamping reads with indices less than i before o completes. The challenge with handling follower failures is providing this guarantee without receiving acknowledgments from all followers.

To achieve this, every follower p only stamps reads with indices less than i while it has an active *lease* for an index $j < i$. A lease for j is a limited-time promise from the leader to p that no RMW stamped with an index greater than j will take effect without p's acknowledgment [15]. This allows p to perform reads like in our

failure-free descriptions, except that p cannot stamp reads with indices smaller than j. Using these leases, the leader can ensure p has stopped stamping reads with indices less than i by waiting for all leases previously granted to p for indices less than i to expire.[6]

We now discuss how to tolerate leader failure. As we have seen, the leader is responsible for granting leases and linearizing RMW operations. These responsibilities must be gracefully transferred between successive leaders to maintain correctness.

Transferring the responsibility of granting leases requires that the previous leader stops granting leases before the new leader begins. This ensures that an old leader cannot grant a lease "behind the back" of the new leader, allowing processes to read a stale version of the state. To avoid this, leader election is done by the enhanced leader service of [9], which guarantees there is at most one leader at any time, and a leader is eventually elected. Once a new leader is elected, it begins granting leases after waiting for all leases from the previous leader to expire. Specifically, it periodically grants leases for the maximum prepared index after it is "initialized", as described next, to the set of active leaseholders. If a process learns it is not in this set, it sends a message to the leader to be added.

Transferring the responsibility of linearizing RMWs is done in a way reminiscent of Paxos. After a new leader ℓ is elected, it is initialized by contacting a majority of processes to ensure that they dismiss prepare messages from all previous leaders and also to determine all half-done operations. ℓ then finishes these operations and begins linearizing new RMWs. A RMW o is linearized in the same way as our failure-free descriptions, except instead of waiting for acknowledgments from all followers, ℓ waits to receive acknowledgments from a majority of processes. This enables future leaders to recover operations linearized by ℓ. Then, ℓ waits until it receives acknowledgments from all active leaseholders or until some *grace* period passes.[7] If the grace period passes, ℓ waits for the leases of all unresponsive active leaseholders to expire and removes them from the set of active leaseholders. This is to ensure a crashed follower only delays one RMW operation. ℓ then sends commit messages for o to all processes. In the fault-tolerant version of PA, ℓ also does this after ℓ's go event occurs. This enables followers who have not received acknowledgments from all active leaseholders to apply o.

8 EXPERIMENTAL EVALUATION

This section answers the following questions experimentally:

(1) What are the effects of asymmetry, α, and $\delta^{\min}$ on latency?
(2) How do latency spikes and follower failures impact PL and PA?
(3) What are the benefits of PL and PA in terms of average latency and throughput compared to existing local read algorithms?

8.1 Implementation & Experimental Setup

Since all but the second part of question (2) is unrelated to fault tolerance, we answer them by comparing failure-free versions of Leader Reads (LR) [8] (described later), Invalidation (INV), Eager Stamping (EAG), Delayed Stamping (DEL), Pairwise-Leader (PL), and Pairwise-All (PA) implemented in Java using gRPC [3]. We then explore the effects that follower failures have on PL and PA using versions of PL and PA that can tolerate follower failures.

All experiments were run on AWS using c5.4xlarge EC2 instances. Each experiment was run five times, and we plotted its average and p99 confidence intervals (but they are barely visible). To answer questions (1) and (2), we use three instances in North Virginia and artificially impose constant message delays using traffic control [4]. To answer question (3), we use two different real-world networks (described in §8.4). Our experiments replicate a no-op state machine, except Figure 15, which replicates a RocksDB key-value store.

When running DEL, we synchronized clocks using NTP [25] and set $\Delta = \overline{D}$. When running PL and PA, we set $\epsilon = 200$ ppm and re-establish markers every 500 ms. Lastly, we configure α for DEL, PL, and PA as discussed in the second experiment of §8.2.

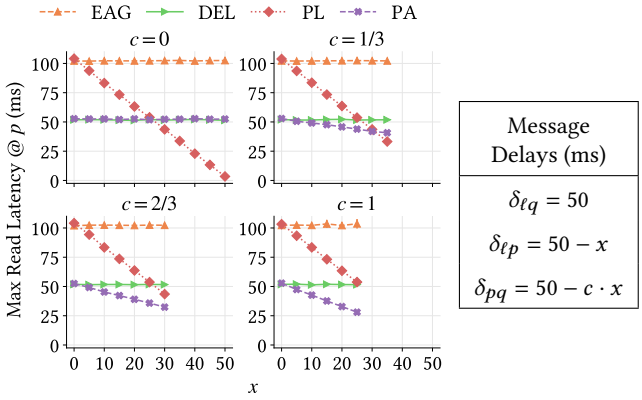


Figure 8: Maximum read latency at process p as the network described by the table on the right becomes more asymmetric.

8.2 Effects of Asymmetry, α, & $\delta^{\min}$ on Latency

To measure the effects of asymmetry, α, and $\delta^{\min}$ on latency, we measure the maximum read and RMW latency during a one-second workload where every process performs a read every millisecond, and the leader performs a RMW every millisecond.

We measure the effects of network asymmetry by using a three-process network where the message delay between the leader ℓ and q is 50 ms, between ℓ and p is $50 - x$ ms, and between p and q is $50 - c \cdot x$ ms where $c \in [0, 1]$. The parameter c controls how asymmetry changes with x. For example, when $c = 0$, p gets closer to ℓ as x increases, but p's distance to q remains equal to 50 ms. Conversely, when $c = 1$, p gets equally closer to ℓ as it does to q as x increases. To isolate the effects of network asymmetry, we vary x such that the diameter is always 50 ms. For the same reason, we do not use lower bounds on message delay, so $\overline{D} = 50$ ms.

Figure 8 shows the maximum read latency at p. As we can see, the read latency of EAG and DEL is *unaffected* by network asymmetry. Read latency with EAG is 100 ms for all c and x since $2 \cdot E_\ell = 100$ ms whereas with DEL it is 50 ms since $\overline{D} = 50$ ms. INV is also unaffected by network asymmetry but is not shown in Figure 8 since all reads block until the experiment finishes. Conversely, the read latency of PL and PA is *affected* by network asymmetry. Read latency with PL is $2 \cdot (50 - x)$ ms since this is $2 \cdot \overline{\delta}_{\ell p}$ whereas with PA it is $\max(50 - x, 50 - c \cdot x)$ ms since this is $\overline{E}_p$. As a result, PL and PA reduce maximum read latency at p by up to 50x and

[6] Leases are traditionally granted using synchronized clocks. Specifically, the leader chooses a time t for which a follower p's lease will expire and sends t to p. The leader then knows that p's lease has expired once $t + \Delta$ elapses on its clock. Alternatively, leases can be granted using the event scheduling primitive's `before` procedure.
[7] To guarantee liveliness, the grace period must be at least $2 \cdot \delta^{\max}$ (defined in §4.3).

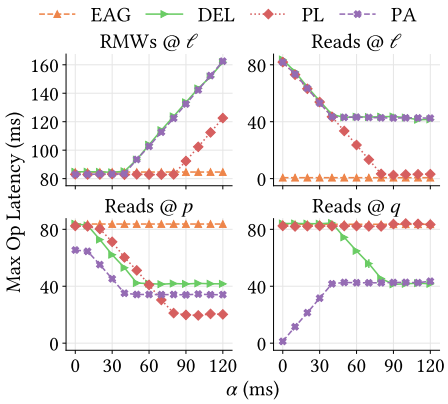


Figure 9: Maximum operation latency as α is varied from 0 to 120 ms.

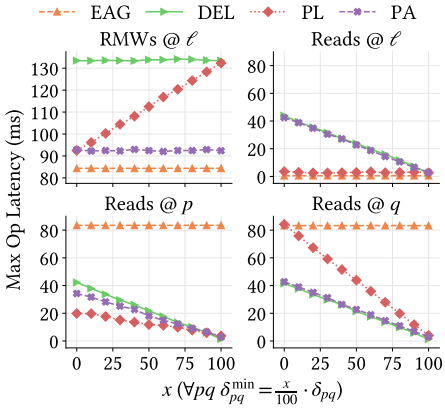


Figure 10: Maximum operation latency as $\delta_{pq}^{\min}$ is varied from 0 to 100% of δ_{pq}.

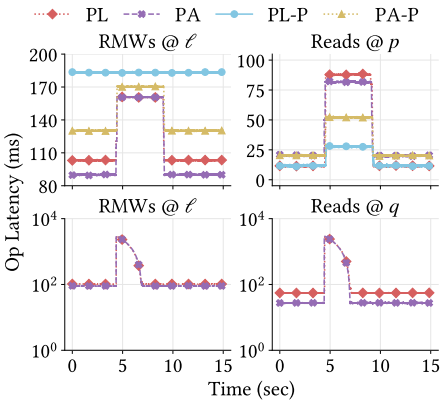


Figure 11: Latency during a network spike (top) and p's failure (bottom).

2x compared to the best existing algorithm, respectively. Finally, notice that the results in Figure 8 nearly match our analysis even though $\epsilon = 200$ ppm while our analysis assumed that $\epsilon = 0$. This is ultimately because markers are re-established every 500 ms.

We measure the effects of α by varying it from 0 to 120 ms in the three-process network shown in Figure 1. Furthermore, to isolate the effects of varying α, we do not use lower bounds on message delay, so $\overline{D} = 40$ ms. Clockwise, Figure 9 shows the RMW latency at the leader ℓ and the read latency at ℓ, p, and q. We include EAG, although it doesn't use α (the same value is plotted for all α).

As shown in Figure 9, α controls the trade-off between RMW and read latency for DEL, PL, and PA. Specifically, RMW latency is flat up until some point, followed by linearly increasing. Read latency has the opposite behavior; it linearly decreases until some point, followed by plateauing. Notice that RMW latency begins to increase when read latency starts to plateau; this instantaneous point is the "sweet spot" for α because smaller values increase read latencies, and larger values increase RMW latencies. However, since this sweet spot is instantaneous, it is impossible to hit it in practice reliably. So, as Figure 9 shows, read latency is reliably minimized when α is at least 90 ms in DEL, 90 ms in PL, and 50 ms in PA. This is 10 ms higher than each algorithm's sweet spot of $3 \cdot E_\ell - \Delta$ for DEL, $2 \cdot E_\ell + E_\ell^{\min}$ for PL, and $E_\ell + D^{\min}$ for PA. Since these algorithms are designed to minimize read latency, we configure α 10 ms higher than these sweet spots throughout our evaluation.

We measure the effects of $\delta^{\min}$ by setting $\delta_{pq}^{\min} = \frac{x}{100} \cdot \delta_{pq}$ for all p and q and varying x from 0 to 100. This experiment is run in the network shown in Figure 1, so $\overline{D} = 40 \cdot (1 - \frac{x}{100})$ ms.

As Figure 10 shows, read latency at all processes with DEL, PL, and PA linearly decreases towards 0 as x increases towards 100. This is expected because our analysis of DEL, PL, and PA showed that their worst-case read blockage times are all linearly dependent on $\delta^{\min}$. Conversely, EAG provides constant read latency as it cannot take advantage of known lower bounds on message delays. Also, notice that the RMW latency with PL linearly increases with x. This is because with PL, the leader's go events are scheduled further in the future as x increases, so its RMW latency also increases. Conversely, with PA and DEL, the leader's go events remain the same irrespective of x, and hence, so does their RMW latency.

8.3 Latency Spikes & Follower Failures

To measure the impact of latency spikes and follower failures on PL and PA, we measure RMW and read latency during the same workload as the last section over fifteen seconds.

We measure the impact of latency spikes in the three-process network shown in Figure 1 by configuring message delays as shown in Figure 1 in the first and last five seconds and by doubling them in the middle five seconds. Moreover, we use the speed of light propagation times shown in Figure 1 as the message delay lower bounds. Since our configuration of α depends on message delays, we compare two different versions. PL and PA configure α *optimistically* using the message delays in the first and last five seconds. In contrast, PL-P and PA-P configure α *pessimistically* using the message delays in the middle five seconds. The top of Figure 11 shows the RMW latency at ℓ and the read latency at p.

As we can see at the top of Figure 11, the read latency at p is the same for both versions of each algorithm in the first and last five seconds. However, during the middle five seconds, the pessimistic versions provide read latency at p equal to our analysis, while the optimistic versions provide higher read latency. This is because the pessimistic versions are always on the read latency plateau we saw in Figure 9, while the optimistic versions are only on it during the first and last five seconds. However, the downside of the pessimistic versions is that they provide higher RMW latency than the optimistic versions. So, to get the best of both versions, the optimistic versions would ideally increase their configuration of α during the middle five seconds. This can be done by integrating the status mechanism of [6] into the Pairwise algorithms.

We measure the impact of follower failures by manually crashing p after approximately 5 seconds in the same network as the last experiment. The bottom of Figure 11 shows the RMW latency at ℓ and the read latency at q in log scale. In this experiment, the grace period is one second, and leases are granted for two seconds.

As we can see in the bottom of Figure 11, after p fails, RMW latency at ℓ and the read latency at q peaks at three seconds, i.e., the sum of the grace period and the lease length as expected. Otherwise, before p fails and after p's lease is revoked, ℓ's average RMW latency is 103 ms with PL and 91 ms with PA, whereas q's average read latency is 55 ms with PL and 27 ms with PA.

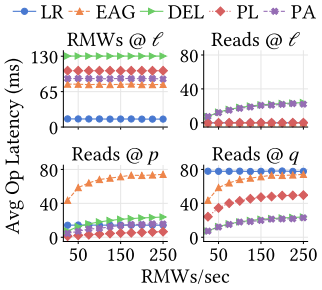


Figure 12: Average operation latency as the number of RMWs per second is varied from 25 to 250.

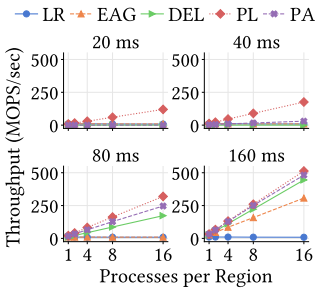


Figure 13: Read throughput with different RMW intervals as the number of processes per region is varied.

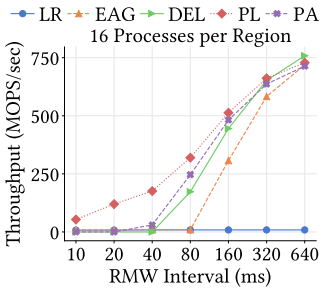


Figure 14: Read throughput as the interval between consecutive RMWs is varied from 10 to 640 ms.

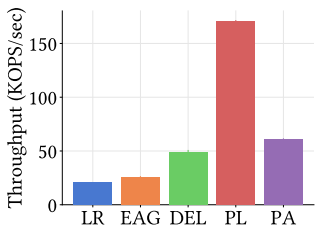


Figure 15: Throughput on a distributed version of RocksDB during concurrent YCSB updates and reads.

8.4 Performance in Real-World Networks

We measure the average latency benefits of PL and PA by performing x RMWs per second at the leader ℓ and by performing a read every millisecond at each process for 10 seconds. This experiment is run across the three AWS regions shown in Figure 1. In this network, $\overline{D}$ is 27 ms, and α is configured to 103 ms for DEL, 103 ms for PL, and 63 ms for PA. Clockwise, Figure 12 shows the average RMW latency at ℓ and the average read latency at ℓ, p, and q.

As we can see in Figure 12, the average read latencies for all local read algorithms converge to their worst-case read blockage times as x increases. This is because as x increases, the likelihood of a read blocking increases. Figure 12 is also the first time we see LR: an optimization to leader-based state machine replication algorithms that forwards all reads to the leader, who then performs them immediately and returns the result [8]. As we can see, LR achieves the lowest RMW latency since only a majority of processes need to acknowledge each prepare message. The cost of this, however, is that it cannot perform reads locally at any follower. As a result, for small values of x, LR's average read latency at p is significantly higher than PL's, and its average read latency at q is significantly higher compared to all local read algorithms. Moreover, LR's read-throughput is considerably lower, as we will see next.

We measure the read-throughput benefits of PL and PA by varying the interval between RMWs from 10 to 640 ms and the number of processes per region (x) from 1 to 16. This experiment has between 1 and 16 processes in the Montreal, North Virginia, North California, Frankfurt, and Stockholm AWS regions. The leader is in Montreal. In this experiment, the leader performs a RMW every 10 to 640 ms, and each process performs reads back-to-back for 1 minute. Moreover, we use the speed of light propagation times as the known lower bounds on message delays, and so in this network, $\overline{D}$ is 58 ms. Furthermore, α is configured to 108 ms for DEL, 134 ms for PL, and 92 ms for PA. Clockwise, Figure 13 shows the read throughput when the RMW interval is 20, 40, 80, and 160 ms as x varies from 1 to 16. Moreover, Figure 14 shows the read throughput when $x = 16$ as the RMW interval varies from 10 to 640 ms.

As we can see in Figure 13, PL provides significantly higher read-throughput than all existing algorithms for small RMW intervals. Furthermore, this improvement scales linearly with the number of processes per region. Concretely, PL and PA improve throughput by up to 19.4x and 3.3x, respectively, compared to the best existing algorithm (when the interval is 40 ms and $x = 16$). These results are expected: since each process performs read operations back to back, a lower worst-case read operation latency translates to more free time to read the state without blocking. This is why in Figure 14, as the RMW interval increases, the gap between all algorithms for local reads shrinks until they coverage at $x = 640$ ms. The exception is LR, which cannot perform reads locally at any follower.

We also evaluate these algorithms in the context of a distributed key-value store. Like prior works [14, 30], we use each algorithm as a shim layer to implement a distributed version of RocksDB. We first ran a YCSB workload of 1000 keys, in which each process ran 16 closed-loop clients performing x% updates and $100-x$% reads for one minute. This experiment was conducted with four processes per region in the same five-region network as the last experiment. In this experiment, the average latency for all local read algorithms linearly increased from ≈ 0 ms at $x = 0$ to the average RMW latency across all processes at $x = 100$. However, this workload does not stress these algorithms because each RMW takes around 100 ms, so each client is mostly idle. To stress these algorithms, we split the above workload's clients into two different groups: eight that only perform updates and eight that only perform reads. Figure 15 shows each algorithm's throughput in this workload.

As shown in Figure 15, LR, EAG, DEL, PL, and PA perform 20.4, 25.3, 47.3, 170.4, and 59.7 KOps/sec, respectively. This is a 3.6x and 1.2x improvement for PL and PA compared to the best existing algorithm, respectively. We note that throughput drops compared to the last experiment for two main reasons: this experiment performs significantly more RMWs, and reads in RocksDB take microseconds while reads in our no-op state machine take nanoseconds.

9 CONCLUSION

We first showed that in periods where message delays are fixed and all processes are non-faulty, the worst-case read blockage time at every follower with all existing linearizable local read algorithms is at least $\overline{D}$ when deployed on commodity hardware. We then presented Pairwise-Leader and Pairwise-All and showed that in the same periods, their worst-case read blockage time at every process p is $2 \cdot \overline{\delta}_{p\ell}$ and $\overline{E}_p$, respectively, which is below $\overline{D}$ at well-located followers. Lastly, we showed experimentally that these new algorithms reduce worst-case read latency by up to 50x and increase read throughput by up to 19.4x compared to existing algorithms.

REFERENCES

[1] [n. d.]. https://aws.amazon.com/. Accessed: 2025-05-19.

[2] [n. d.]. http://rocksdb.org.. Accessed: 2025-05-19.

[3] [n. d.]. https://grpc.io/. Accessed: 2025-05-19.

[4] Werner Almesberger. 1999. Linux network traffic control–implementation overview. In *5th Annual Linux Expo*. 153–164.

[5] Jason Baker, Chris Bond, James C Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing scalable, highly available storage for interactive services.. In *CIDR*, Vol. 11. 223–234.

[6] Changyu Bi, Vassos Hadzilacos, and Sam Toueg. 2022. Parameterized algorithm for replicated objects with local reads. *arXiv preprint arXiv:2204.01228* (2022).

[7] Saâd Biaz and Jennifer L Welch. 2001. Closed form bounds for clock synchronization under simple uncertainty assumptions. *Inform. Process. Lett.* 80, 3 (2001), 151–157.

[8] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. 398–407.

[9] Tushar D Chandra, Vassos Hadzilacos, and Sam Toueg. 2016. An algorithm for replicated objects with efficient reads. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*. 325–334.

[10] 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*. 143–154.

[11] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.

[12] Flaviu Cristian and Christof Fetzer. 1999. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed systems* 10, 6 (1999), 642–657.

[13] Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. 2020. State-machine replication for planet-scale systems. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–15.

[14] Jinkun Geng, Anirudh Sivaraman, Balaji Prabhakar, and Mendel Rosenblum. 2022. Nezha: Deployable and High-Performance Consensus Using Synchronized Clocks. *Proceedings of the VLDB Endowment* 16, 4 (2022), 629–642.

[15] Cary Gray and David Cheriton. 1989. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *ACM SIGOPS Operating Systems Review* 23, 5 (1989), 202–210.

[16] Haryadi S Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J Eliazar, Agung Laksono, Jeffrey F Lukman, Vincentius Martin, et al. 2014. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the ACM symposium on cloud computing*. 1–14.

[17] 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.

[18] Owen Hilyard, Bocheng Cui, Marielle Webster, Abishek Bangalore Muralikrishna, and Aleksey Charapko. 2023. Cloudy Forecast: How Predictable is Communication Latency in the Cloud? *arXiv preprint arXiv:2309.13169* (2023).

[19] MR Siavash Katebzadeh, Paolo Costa, and Boris Grot. 2020. Evaluation of an infiniband switch: Choose latency or bandwidth, but not both. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 180–191.

[20] Antonios Katsarakis, Vasilis Gavrielatos, MR Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. 2020. Hermes: A fast, fault-tolerant and linearizable replication protocol. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 201–217.

[21] Leslie Lamport. 2001. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001), 51–58.

[22] Tanakorn Leesatapornwongsa, Jeffrey F Lukman, Shan Lu, and Haryadi S Gunawi. 2016. TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *Proceedings of the twenty-first international conference on architectural support for programming languages and operating systems*. 517–530.

[23] Christoph Lenzen, Thomas Locher, and Roger Wattenhofer. 2010. Tight bounds for clock synchronization. *Journal of the ACM (JACM)* 57, 2 (2010), 1–42.

[24] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, et al. 2020. Sundial: Fault-tolerant clock synchronization for datacenters. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*. 1171–1186.

[25] David L Mills. 1985. *Network time protocol (NTP)*. Technical Report.

[26] Iulian Moraru, David G Andersen, and Michael Kaminsky. 2014. Paxos quorum leases: Fast reads without sacrificing writes. In *Proceedings of the ACM Symposium on Cloud Computing*. 1–13.

[27] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX annual technical conference (USENIX ATC 14)*. 305–319.

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

[29] Nathan VanBenschoten, Arul Ajmani, Marcus Gartner, Andrei Matei, Aayush Shah, Irfan Sharif, Alexander Shraer, Adam Storm, Rebecca Taft, Oliver Tan, et al. 2022. Enabling the next generation of multi-region applications with cockroachdb. In *Proceedings of the 2022 International Conference on Management of Data*. 2312–2325.

[30] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan R. K. Ports, Ion Stoica, and Xin Jin. 2019. Harmonia: Near-Linear Scalability for Replicated Storage with In-Network Conflict Detection. *Proceedings of the VLDB Endowment* 13, 3 (2019), 2203–2215.