The LAW theorem: Local Reads and Linearizable Asynchronous Replication

Antonios Katsarakis[**]
Vasilis Gavrielatos[*]
Huawei Research
first.last@huawei.com

Emmanouil Giortamis[*]
Pramod Bhatotia
TU Munich
first.last@tum.de

Aleksandar Dragojevic
OpenAI
aleksandar.dragojevic@gmail.com

Boris Grot
University of Edinburgh
boris.grot@ed.ac.uk

Vijay Nagarajan
University of Utah
vijay@cs.utah.edu

Panagiota Fatourou
FORTH ICS and University of Crete
faturu@csd.uoc.gr

ABSTRACT

Distributed datastores underpin highly concurrent, read-intensive applications, ensuring consistency, availability, and performance. They use crash-tolerant protocols to replicate data and endure replica server crashes. To ensure safety and meet the performance demands, replication must support high-throughput, strongly consistent (i.e., linearizable) reads without assuming any synchrony. However, existing protocols either ❶ relax consistency, or provide linearizable reads that are ❷ fully asynchronous but remote (involving multiple replicas), or ❸ local but require synchrony.

This work explores the tradeoffs between consistency, asynchrony, and performance in crash-tolerant protocols, and proves that *in linearizable asynchronous read/write registers tolerating a single crash, no reads can be local*. Building on this, we introduce *almost-local reads* (ALRs), a new abstraction that ensures crash tolerance and linearizability under asynchrony. While ALRs have slightly higher latency than local reads, they remain lightweight, with computation and network costs close to single-node reads.

We present two simple yet effective ALR schemes that enhance protocols across all three categories. For protocols with local reads, ALRs address consistency or synchrony issues with minimal throughput loss. In asynchronous linearizable protocols, they improve performance without compromises. Our evaluation shows that ALR-enhanced ZAB and Hermes achieve within 2% and 5% of their original throughput in 95% reads while ensuring linearizability under asynchrony. On Raft, ALRs deliver over 2.5× higher throughput without compromising consistency or asynchrony.

PVLDB Reference Format:
Antonios Katsarakis, Emmanouil Giortamis, Vasilis Gavrielatos, Pramod Bhatotia, Aleksandar Dragojevic, Boris Grot, Vijay Nagarajan, and Panagiota Fatourou. The LAW theorem: Local Reads and Linearizable Asynchronous Replication. PVLDB, 18(9): 2831-2845, 2025. doi:10.14778/3746405.3746411

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://law-theorem.com/.

Proceedings of the VLDB Endowment, Vol. 18, No. 9 ISSN 2150-8097.
doi:10.14778/3746405.3746411

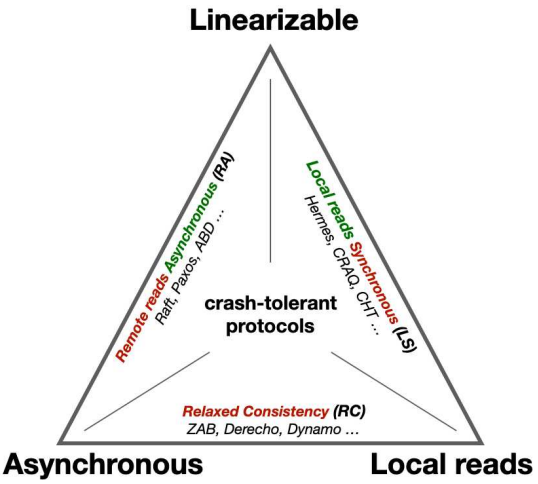

Figure 1: *Three-way tradeoff of crash-tolerant protocols.*

1 INTRODUCTION

Today's online services and cloud applications rely on in-memory high-performance datastores[1], such as key-value stores, for storing and accessing their data. These applications are characterized by read-dominant access patterns and concurrent requests [10, 32, 114]. Thus, datastores must provide high throughput to meet the performance demands of modern applications and offer high availability, as they are deployed on fault-prone commodity hardware [17].

Data replication is a core feature of high-performance and resilient datastores. Data must be replicated across multiple nodes (i.e., servers) to increase throughput because a single node often cannot keep up with the load [24]. Replication is also necessary to guarantee that a node crash does not render the dataset inaccessible.

Maintaining the replicas consistent, to ensure that the services running on the datastore operate correctly, is a challenge, especially in the presence of crashes. A *crash-tolerant replication protocol* is responsible for keeping the replicas of a datastore consistent – even when crashes occur – by determining the necessary actions to execute reads and writes. Several crash-tolerant protocols favor performance by relaxing consistency (*RC* protocols). As such, their reads may return stale values, leading to nasty surprises for both clients and developers [89, 124]. There exist however, protocols that

[*]The two authors contributed equally to this work.
[*]This work started when the authors were at the University of Edinburgh.
[1]We use the term datastore to encompass in-memory storage systems within a local area network with an API that includes reads and writes to objects.

offer strongly-consistent (i.e., linearizable) reads, which are more desirable for correctness and programmability.

Crash-tolerant protocols can be synchronous or asynchronous, based on the timing model they rely on to ensure consistency. Synchronous protocols, which depend on bounded processing and communication delays, are easier to design. However, in the real world, distributed datastores are deployed over complex software stacks and virtualization layers [18, 84]. Consequently, the network and compute nodes of a distributed datastore experience asynchrony and other timing anomalies, which may lead to timing violations and compromise the safety of synchronous protocols. To tolerate such timing violations, safer protocols adopt the *asynchronous* model where there are no timing assumptions, implying that processing and communication delays can be arbitrary.

Existing crash-tolerant replication protocols that afford linearizable reads fall in two categories; Local-Synchronous (*LS*), and Remote-Asynchronous (*RA*). Protocols in the LS category offer cheap linearizable reads that complete locally on a replica (i.e., without inter-replica communication), as in Hermes [66], but these protocols assume a synchronous model [42] (e.g., exploiting lease mechanisms). In contrast, protocols that fall into the RA category such as Raft [100], Paxos [75], and ABD [11, 91] are safe under asynchrony but each and every read mandates expensive inter-replica communication and processing costs on multiple replicas. As Schwarzmann and Hadjistasi explain [54, 55], it is important to study the feasibility of crash-tolerant implementations that support linearizable local (*zero-delay* in their terminology) reads under asynchrony.

There exist fundamental theoretical results related to asynchronous replication, including the seminal FLP result [45] and the CAP theorem [22, 50], but neither suffice to answer the above as both fall short in examining the performance of reads. We detail these and other relevant results in § 7. To address the existing gap in understanding, this work explores the fundamental three-way tradeoff of crash-tolerant protocols, revealing a tension between consistency, performance, and the time assumptions of the setting (also depicted in Figure 1). In particular, we prove that *in any linearizable asynchronous read/write register implementation that tolerates even a single crash (without blocking reads or writes), no reads are local.*

We also observe that the performance aspect of this tradeoff affects the latency but not necessarily the throughput of reads. Thus, asynchronous linearizable reads need not be as costly as in existing (RA) protocols, where each read incurs network and computation costs to remote replicas.

Capitalizing on this insight and the high volume of concurrent requests in modern read-intensive applications, we introduce *almost-local reads* (ALRs), a technique that affords low-cost reads in a linearizable and crash-tolerant manner under asynchrony. In short, ALRs batch read requests with a twist. Unlike traditional batching, all reads in an ALR-batch are executed against the local replica of a server, and only a lightweight *sync* operation per batch involves remote replicas. The sync incurs only a small network and computation cost, regardless of the batch size. Moreover, it can sometimes be elided as existing writes can act as implicit syncs(§ 4.3). As a result, ALRs incur little or no extra network and processing costs to remote replicas, thus achieving the performance of local reads while offering linearizability under asynchrony. We detail two simple ALR schemes; an eager and a lazy one. Despite their simplicity, these schemes can benefit a variety of protocols by eliminating either consistency or asynchrony relaxations, or by boosting performance. In short, the contributions of this work are as follows:

- We classify crash-tolerant replication protocols based on three features: linearizability, read locality, and asynchrony, highlighting the tradeoff between consistency, performance, and time models. (§ 2). We then show that in any asynchronous linearizable register implementation that tolerates even a single crash without blocking reads or writes, no reads are local. Finally, we examine the boundaries of this result, showing its tightness. (§ 3)
- We introduce two *almost-local read* (ALR) schemes, an eager and a lazy one, which allow reads with computation and network overhead close to that of local reads. These schemes can be applied to a wide range of crash-tolerant protocols for high throughput reads without sacrificing asynchrony or linearizability. (§ 4)
- We implement ALR-enhanced variants of three state-of-the-art protocols, one per class (RC, LS, RA) and evaluate their performance over RDMA. ALRs increase up to 2.6× the throughput of Raft (RA) in read-intensive workloads without compromises. The ALR-enhanced ZAB (RC) and Hermes (LS) protocols on 95% reads show only marginal throughput reduction of 2% and 5% and minimal latency increase, but are also providing linearizability under asynchrony. Notably, these benefits are over baselines that already heavily batch requests to boost throughput. (§ 6)

2 BACKGROUND

2.1 Datastores and workload characteristics

Modern distributed datastores keep the data in memory and serve as the backbone for many of today's data-intensive services, including e-commerce and social networks. These workloads are characterized by a high volume of concurrent requests and read-dominant accesses [10, 24, 115]. To satisfy the demands, datastores must offer high throughput – especially on reads. This work focuses on datastores deployed within a local area network (e.g., a datacenter).

2.2 Replication and consistency

Datastores partition data into *shards* and replicate shards to ensure crash tolerance. A crash-tolerant replication protocol maintains consistency across a shard's replicas. The *replication degree* (number of replicas) balances cost and fault tolerance: higher degrees improve resilience but increase deployment costs. A replication degree of 3 to 7 is considered a good balance for safety and cost [58].

When data are replicated, consistency must be enforced. While weak consistency can be leveraged to boost performance [82, 86, 112], it can also lead to nasty surprises when developers or clients attempt to reason about the system's behavior [124]. For this reason, we mainly focus on sequential consistency and linearizability. Sequential consistency mandates that reads and writes from each client appear to take effect in some total order consistent with the order in which they were issued [73]. In addition to sequential consistency's constraints, linearizability mandates that each request appears to take effect instantaneously at some point between its invocation and response [56]. In practice, linearizability is preferable to sequential consistency because it is compositional: when all shards enforce linearizability, then the datastore enforces linearizability. This does not hold for sequential consistency.

2.3 Synchrony and asynchrony

In distributed datastores, a *synchronous* model [42] assumes bounded processing and communication delays. These assumptions allow protocols to handle unresponsive replicas via end-to-end timeouts. However, modern datastores cannot always meet end-to-end time properties due to their complex software stacks (e.g., network protocols, garbage collection, and virtualization layers), each with unpredictable time behavior [15, 84]. Although such systems may have predictable timing in the typical case, load spikes can easily translate into delays that violate the model's timing assumptions [4].

Because the synchronous model fails to represent real-world distributed datastores, even protocols correct under the model can violate consistency in practice. For instance, consider a synchronous protocol that relies on leases for safety; if the timing assumptions cease to hold, a server might falsely believe it still holds a lease that has expired from the view of other participants. Consequently, it can subtly compromise the protocol's safety.

Many replication protocols adopt the *asynchronous* model to resolve the safety issues in the synchronous model. In the asynchronous model, there are no timing guarantees—processing and communication delays are arbitrary. This ensures high safety because if a system is safe with arbitrary communication delays, it is also safe with shorter delays.

However, dropping time assumptions makes the design of high-performance protocols challenging. Concepts exploited for performance that are related to time (e.g., leases or global clocks) across distributed nodes cannot be utilized in asynchronous protocols. Finally, as the FLP result [45] indicates, asynchrony also renders some problems impossible to solve.

2.4 Crash-tolerant replication protocols

We classify replication protocols able to deal with node replica crashes into the following three categories.

❶ **Local Reads under Relaxed Consistency (RC).** Replication protocols of this class offer local reads and tolerate crashes under asynchrony but relax linearizability [33, 48, 59, 61, 68, 72, 103, 110, 122], thus affecting the correctness and programmability. We say these protocols fall into the Relaxed Consistency (RC) category of crash-tolerant replication protocols. While there exist a plethora of protocols that relax consistency, including protocols that offer very weak models such as eventual consistency [25], in this work, we primarily focus on those relaxing linearizability the least by providing sequential consistency (e.g., as in ZAB [61] and Derecho [59]).

❷ **Linearizable Local Reads under Synchrony (LS).** In this class, crash-tolerant linearizable protocols relax asynchrony to prioritize read performance. Typically, this is achieved by mandating some global timing assumptions to implement leases [51]. This approach enables efficient *local reads* (i.e., reads returning the local value of a replica without the replica delivering a message within the read's invocation and response), which guarantees linearizability despite the absence of replica coordination. Leases are used in either majority- or membership-based protocols [66] to temporarily maintain a configuration. In majority-based protocols, leases protect a stable leader configuration allowing the leader to serve local reads while the lease has not expired [27, 30, 128]. In membership-based protocols, leases protect the membership configuration, which specifies all *live* nodes, enabling local reads on one (e.g., as in Primary-backup [7]) or on all replicas of the configuration (e.g., as in Hermes [66] or CRAQ [117]).

In both approaches, leases temporarily "lock" the configuration, enabling linearizable local reads that greatly improve performance. However, leases require strong synchrony assumptions. The system's clocks, process speeds, and message delays must always operate within, and not diverge from, certain bounds (i.e., adhere at least to the partially-synchronous model [42]).

❸ **Linearizable Remote reads under Asynchrony (RA).** There exist several protocols that ensure linearizability and can tolerate crashes in the asynchronous setting – i.e., where clocks, processors, and messages can all operate at arbitrary speeds. Note that this class of protocols also includes *indulgent* protocols [52], which are safe under asynchrony but may not always guarantee progress. Protocols in this class either directly implement an atomic register (e.g., as in the seminal ABD [11]) or they leverage a consensus protocol to implement the state machine replication [109] (SMR) approach to serve reads and writes (e.g., as in Raft [100] and Paxos [75]).

Existing protocols in this class either constrain reads to be executed only by a designated leader replica [100] or require costly coordination for each linearizable read [12, 61, 75]. In the latter case, most protocols treat reads as expensive as writes or require one or more round-trips to a majority of replicas, incurring network and processing costs for all participating replicas on every read.

3 THE L^{2}AW IMPOSSIBILITY

Ideally, a crash-tolerant system i.e., a replication protocol, should offer strongly-consistent (linearizable) local reads from all replicas in the asynchronous setting, while being able to tolerate the crash of any minority of replicas (without indefinitely blocking reads or writes). However, we prove the *L^{2}AW impossibility*, which states that *in any **L**inearizable **A**synchronous read/write register implementation that tolerates even a single crash (**W**ithout blocking reads or writes), no reads are **L**ocal.*

Intuitively, in an asynchronous system, a replica cannot reliably determine whether another replica has crashed or is simply slow. To maintain linearizability, a read operation must reflect the latest writes, which often requires coordination between replicas. However, only allowing a replica to perform a local read means it might miss updates from other replicas, leading to inconsistent or stale results. This inherent uncertainty and the need for coordination make it impossible to guarantee both crash tolerance and even *a single* strongly consistent local read simultaneously. Next, we proceed with the model specification and the L^{2}AW impossibility.

3.1 Model and definitions

System model. We model a distributed system that consists of a set of N server nodes, each hosting a replica process (from now on *replicas*). The replicas communicate over an asynchronous network by sending and receiving messages. A set of client processes (clients) issue requests to server processes (servers) that store data replicas. Replicas are modeled as deterministic state machines: in every computation *step*, they may perform some deterministic local computation, and in every communication step, they may deliver a

message or send a message to another replica, atomically. An *event* is defined as a computation or communication step performed by any replica. The replicas fully replicate a *binary* register and execute reads and writes over the register when serving client requests.

Clients. Client processes (from now on *clients*) issue read and write requests that replicas serve. We do not make any assumptions regarding their location; clients could be either co-located with one (but any) of the replicas as in the ABD algorithm and state-of-the-art datastores [38, 63], or live on separate nodes as in Chain Replication [123]. As in several recent systems and protocols, in our model, clients maintain up to one connection with one replica at a time, i.e., clients do not multicast to servers [28, 38, 61, 66, 100]. Note that reducing client-server connections is critical for the scalability of replicated datastores [99]. For the remainder of this work, we do not consider clients as part of the algorithm; they are off-path for any operations mentioned, and we omit them for simplicity.

Configuration. We represent the system configuration C, as a vector: $C = (s_1, ..., s_N, \rho)$, where $s_1, ..., s_N$ are the states of N replicas $\{R_1, ..., R_N\}$ and ρ is the register's value. We define an execution *fragment* as sequence of the form $\{E_k, C_k, E_{k+1}, C_{k+1}, ...\}$ where each C_k is a configuration and each E_k is an event. An *execution* is an execution fragment starting from an initial configuration C_0.

Linearizable register. We consider a typical linearizable (i.e., *atomic* [74]) binary multi-writer multi-reader (MWMR) register [106] that is fully replicated across N replicas. Note that our result trivially holds in a single reader (writer) setting, which cannot tolerate the crash of sole reader (writer) and ensure progress on reads (writes).

Operations. The execution interval of a read/write operation op in an execution e starts with the invocation event of op (by a replica R) and ends with the response of op (by R). If the response does not exist, we say that op (and R) is *active* in e, and the execution interval is defined by the suffix of e starting with op's invocation. Each replica may have one active operation at a time. Two operations are *concurrent* if the invocation event of one of them occurs between the invocation and the response events of the other. An operation runs *solo* if it is not concurrent with any other operations.

Asynchronous reliable network. The replicas communicate via a fully-connected, bidirectional, point-to-point, and reliable network (i.e., without message losses or network partitions). A reliable network makes our result also valid for an unreliable network, thus stronger. The network is asynchronous [45], so there are no bounds in transmission delays, but the messages are eventually delivered.

Single crash. We say a replica R performs a *crash* event c in an execution e, if it stops executing computation and communication steps after c. We call replicas that do not crash in e as *correct*. An execution without a crash is *crash-free*. For a stronger result, we assume that up to one (but any) replica R may crash and do not consider recovery or Byzantine faults [90, 104].

Liveness. A read or a write invoked by a correct replica should eventually terminate[2] (despite asynchrony or one possible crash).

Linearizable read. Consider any linearizable execution e. Let L be the set of read operations in e, r a read operation in L, and w a write operation whose linearization point is the last preceding r's linearization point in the linearizable order of e. In this setting, r is linearizable if r returns the value written by w.

Local read. We define a local read as a read returning the local value of a replica R, when no messages are delivered or sent by R, between the read's invocation and response.

3.2 The proof of the impossibility

Sketch and proof. We make two key observations that hold in the specified model and lead to the proof of L^2AW. Informally, the first states that, to tolerate faults under asynchrony, a replica performing an operation op must not rely on a specific replica to complete the op. The second states that a local read executed by a replica may appear to run solo and not reflect the effects of a previously terminated write issued from another replica.

Building on both insights, we prove the L^2AW impossibility by way of contradiction using indistinguishable executions. In particular, we assume that there is a linearizable implementation I of a binary MWMR register in the specified model that produces one or more linearizable local reads. We then show that for each execution produced by I, and for each linearizable local read r contained in it, we can construct an indistinguishable execution from the perspective of the node executing r where r is not linearizable – hence violating the specification of I and leading into a contradiction.

The intuition behind our first Lemma derives from the fact that a crashed replica is indistinguishable from an arbitrarily slow one in the asynchronous message-passing setting [45]. Thus, a read or write operation op invoked by a replica R does not have to wait for a message sent by a specific replica R' to terminate, since R' may have crashed and never respond.

Lemma 1

Fix any finite crash-free execution e produced by I, and let C be the final configuration of e. Assume that some replica R is inactive in C (i.e., it does not have any active operation at C). Let S be the set of all active operations at C. Then, in every long-enough extension e' of e that does not contain any step from R, the following claims hold:

(a) Every operation $op \in S$ invoked by a correct replica, op terminates in e'.
(b) If C is idle ($S = \emptyset$), then each operation op invoked by a correct replica $R' \neq R$ starting from C terminates in e'.

Proof. We add a failure event c for replica R at the end of e (which is finite and crash-free) and let C' be the resulting configuration. Fix any replica $R'' \neq R$. If R'' is active in S, let op be the active operation of R'' at C (and C'). Otherwise, let op be a new operation initiated by R'' started from C'. In an extension e' of e, in which all replicas other than R (including R'') take enough steps, the liveness condition implies that op will terminate. This is true for every R''. The above implies that the claims of the lemma hold. (In an execution that is exactly the same as e' but R does not crash and simply takes no steps, all operations in S would again terminate.) □

[2] Throughout the paper we use *complete* and *terminate* interchangeably.

Lemma 2

Let e be any execution produced by a linearizable implementation I and let r be any local read, executed by some replica R. There is a crash-free execution e' in which:

(a) r is not concurrent with any operation, and
(b) r has the same response in both e and e'.

Proof. We construct e' by using the prefix π of e up until the point C just before r is invoked.

Assume first that π is not crash-free. Let c be the crash event it contains, and let R' be the replica that has crashed. Since R invokes r after π, it follows that $R' \neq R$. To construct e', we remove c from π and let R' receive and process all messages that have been sent to it in π and have not yet been processed. Let β be this execution. If π is crash-free then let $\beta = \pi$.

Let S be the set of active operations at β. Let op be any operation in S. By Lemma 1(a) (applied on β), op must terminate even without R. The same holds for all operations in S. Starting from the configuration after the execution of β, we let all replicas other than R to take steps until all active operations in S terminate. Let M be the set of messages that are sent to R during this execution fragment. We temporarily delay the delivery of these messages in M. Then, we let R to take steps solo until r terminates without invoking any new operation during r's execution. Let e'' be this execution. Note that e'' is indistinguishable to R from π. Finally, we deliver to R the messages contained in M, and let R and all the other replicas take steps until no message is in transit and no operation is active. Let e' be this execution. Thus, R returns the same value for r in π (and thus also in e) and in e'' (and thus also in e'). □

Theorem: L^{2}AW impossibility

Consider any linearizable implementation I of a read/write register in an asynchronous network that tolerates a single replica crash. There exists no execution produced by I which contains a local read.

Proof. Fix an execution e produced by I. Let r be any local read in e executed by a replica R e.g., as in Figure 2. Let e' be the crash-free execution advocating Lemma 2 and let C be the configuration just before the invocation of r in e'. By Lemma 2, for each replica $R' \neq R$, R' does not have an active operation at C (i.e., R' is inactive at C). Let R' be any such replica. Let x be the value that r returns in e'. We construct an execution e'', where R' invokes a $write(x')$ operation W at C, where x' is a value other than x returned by r in e'. Replicas other than R' do not invoke any new operation in e''. Then, all replicas besides R (including R') take steps until W terminates. Lemma 1(b), implies that the execution of W will terminate–even if R takes no steps. Let C' be the configuration at which W has terminated. Next, R takes steps from C', invokes r and runs solo until r terminates. Finally, all replicas take steps to receive all in transit messages. Let π be the prefix of e'' up until the point that the execution of r terminates. By construction, π is indistinguishable from e' to R. Thus, R returns the same value for r in both executions. Recall that r returns x in e'. Since W is the last terminated write prior the invocation of r in e'' and no other operation is active while W and r are executed, r must return x' in e''. Since x was chosen to be different from x', this is a contradiction. □

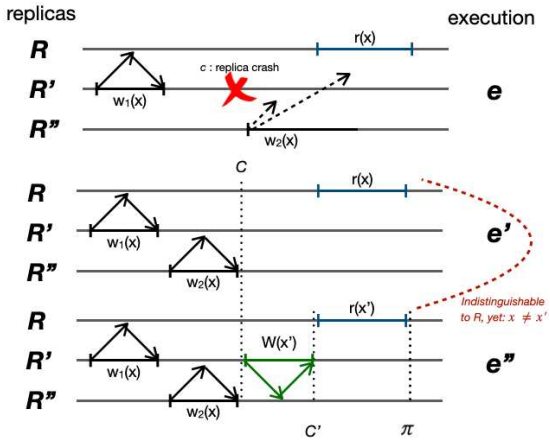

Figure 2: *Example of indistinguishable executions for replica R used in the proof of the L^{2}AW impossibility. Dotted lines denote configurations and execution prefixes. Dashed arrows indicate messages in transit.*

3.3 L^{2}AW tightness

We indirectly prove the tightness of the L^{2}AW impossibility result by showing that there exist crash-tolerant protocols (that have been shown correct) with every combination of two out of the three main properties and by indicating that all three properties are feasible in the absence of crashes. We use the abbreviations *Li*, *Lo*, *A*, and *W* for Linearizability, Local reads, Asynchrony, and crash-tolerance (Without blocking reads or writes), respectively.

> *Lo+A+W (Relaxed consistency):* Our proof hints that protocols that relax the real-time guarantees from the consistency model (e.g., by degrading linearizability to sequential consistency) could be crash-tolerant and afford local reads under asynchrony (e.g., as in ZAB and Derecho [59]).

> *Li+A+W (Remote reads):* As detailed in § 2, there are protocols that tolerate crashes under asynchrony and support linearizable but costly remote reads (e.g., Raft and ABD [91]).

> *Li+Lo+W (Synchrony):* Some crash-tolerant protocols consider a partially-synchronous model, relaxing asynchrony to offer local and linearizable reads via time-based leases (e.g., Hermes and CHT [28]).

> *Li+Lo+A (Non-crash-tolerant):* Without crash tolerance, there exist local reads that are linearizable even under network asynchrony. For instance, the ccKVS protocol [46] cannot tolerate crashes but it does not assume any synchrony for its local linearizable reads.

3.4 Extending the L^{2}AW boundaries

In this section, we scrutinize all key properties of the L^{2}AW impossibility and provide further insights.

Inspecting *Li*: RA-Linearizability. Lev-Ari et al. [80] define a consistency model in-between linearizability and sequential consistency. This model is practical. Although it is weaker than linearizability, it is compositional, unlike sequential consistency. We name a variant of that model Read-Asynchronous Linearizability (RA-Linearizability). RA-Linearizability is the same as Linearizability only that the invocation of a read operation (and thus its

linearization point) can be extended up to the point right after the linearization point of the exact previous operation in the program sequence – iff that previous operation was issued to the same replica. Such reads, which can be linearized in the *past*, can be served locally while tolerating crashes in an asynchronous setting. However, every such locally completing read may return stale data.

Inspecting *A*: partial-synchrony and failure detectors. We believe that works considering a partially-synchronous model [28, 66] have already reached the practical limit of relaxing asynchrony without any other L^{2}AW properties in a typical environment. Nevertheless, it is worth exploring a more abstract concept that makes fewer synchrony assumptions than the above works by defining a new failure detector class [29]. We leave this for future work.

Inspecting *W*: 1) final value. If a crash-tolerant register has an identifiable final value, and that value is observed during a local read to a replica, then the replica may return that value and offer linearizability without consequences. Simply, there is no risk of blocking the system from completing future writes to the register as there will be none. This scenario applies to special register types (e.g., write-once [44]) and it does not invalidate our impossibility.

Inspecting *W*: 2) single writer. Several atomic register protocols provide linearizability under asynchronoy by assuming a single predefined writer replica (also called SWMR protocols). For such protocols, if the stable writer is also a reader it can trivially provide linearizable local reads under asynchrony [96]. Nevertheless, it is apparent that once that single writer crashes the system cannot complete any further writes (i.e., is not live by our definition). Also, note that our proof suffices to show that other readers in such protocols can never guarantee linearizable local reads.

Inspecting *Lo*: almost-local reads. Modern read-intensive applications with numerous concurrent requests would highly benefit if we could accelerate reads while tolerating crashes without relaxing consistency or asynchrony. Thus, the remainder of this work examines the following key question.

Key Question

Can crash-tolerant protocols offer reads almost as cheap as local without sacrificing linearizability under asynchrony?

4 *ALMOST-LOCAL READS* (ALRS)

The L^{2}AW theorem asserts that in a crash-tolerant algorithm, no reads can ever be local under linearizability and asynchrony. Plainly, reads must pay the latency of reaching one or more remote replica servers. However, unlike writes, reads need not alter the state of remote replicas; hence, should be more efficient throughput-wise.

Based on this observation we propose *almost-local reads (ALRs)*, an amortization strategy wherein, only one lightweight remote *sync* operation is used to complete a batch of otherwise locally executed read operations. Crucially, the cost of the *sync* is independent of the size and the contents of the batch. There are two types of *sync*. The first validates if reads executed *eagerly* (before the *sync*) on the local replica are linearizable. The second ensures that reads executed *lazily* (after the *sync*) on the local replica will be linearizable.

As shown in Figure 3, ALRs approximate the performance of local reads, while offering linearizability under asynchrony. When applied to any of the three protocol categories (RA, RC, LS), ALRs

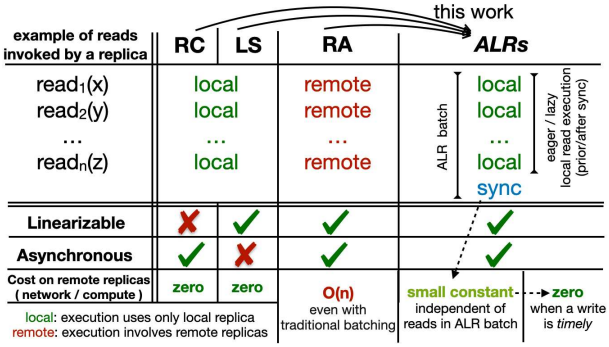

Figure 3: *Operational cost and features for protocols falling in RC, LS, and RA; and when enhanced with ALRs.*

add the missing piece: ❶ they improve throughput for RA protocols (e.g., Raft) ❷ ensure linearizability for RC protocols (e.g., ZAB) and ❸ allow LS protocols (e.g., Hermes) to operate under asynchrony.

ALRs can be applied to a broad set of protocols (e.g., Raft, ZAB, Hermes, Paxos, Chain Replication), but not all of them. Below we detail how we implement ALRs in different protocols and which protocols are amenable to this technique. We start with LS protocols.

4.1 *Eager-ALRs* for Local-Synchronous (LS)

In this section, we exploit ALRs to allow LS protocols operate safely under asynchrony without sacrificing throughput.

Most LS protocols establish a stable *replica configuration* for a time period. A replica is a member of the configuration if it holds a time-based lease. A write cannot complete unless it reaches all leaseholders (i.e., all configuration members). In return, leaseholders exploit this to perform reads locally. However, leases require synchrony to establish a period during which the configuration is stable. Without synchrony, a replica can falsely believe that its lease has not expired and read a stale local value, violating linearizability.

Eager-ALRs. The observation is that we can avoid time-based leases, and thus synchrony, as long as a replica checks that it is still in the configuration after executing a batch of local reads. We call such reads *Eager-ALRs* because they first read the local storage optimistically, and then they perform a *sync*, to ensure the reads were correct. In this case, the sync validates that the configuration during the execution of reads is valid (i.e., is the most recent one). Specifically, a replica *R* executes a batch of Eager-ALRs as follows:

(1) *R* forms a batch of reads and records its local configuration. To construct a batch, a protocol includes pending reads from one or more clients (respecting the request arrival order if necessary). If its configuration permits, it executes the reads against its local replica and buffers the read values.
(2) Then, *R* sends a *sync* message to remote replicas containing the configuration prior to the read execution. Upon receiving this message, remote replicas check if the received configuration agrees with their local (i.e., most recent) configuration, and reply with an Ack or Nack message.
(3) Upon receiving Ack messages from a sufficient quorum of replicas (e.g., a majority guarding the configuration)[3], *R* completes the reads, responding with the buffered values.

[3]For simplicity, we assume that the configuration algorithm is handled by replica servers (as in [66]) and not by external servers.

(4) If R receives a quorum of Nacks, the configuration used to execute its reads locally is not the most recent. Thus, to avoid inconsistencies, R falls back to the original protocol and initiates its recovery (e.g., to renew its configuration).

Correctness. After reading the locally stored values, the replica validates that it is still a member of the configuration, and is thus guaranteed to have received all completed writes. Thus, reads observe completed writes, upholding linearizability but without needing time-based leases or synchrony.

Performance. Adding the *sync* message to the read algorithm results in a negligible throughput cost, as it is amortized among all reads in the batch. This is corroborated by our evaluation in § 6. However, Eager-ALRs have higher latency than local reads, due to the network exchange on their critical path. This is the fundamental cost of linearizability under asynchrony. In § 4.3, we discuss how we mitigate this latency, and in § 6, we show that the extra latency is just a few microseconds (in a modern local area deployment).

Applicability. Eager-ALRs are applicable to any protocol that keeps a stable configuration and exploits time-based leases to read locally while offering linearizability. This includes Hermes, Chain Replication, CRAQ, and Primary-backup.

Summary. Eager-ALRs allow LS protocols to offer linearizability under asynchrony. Inevitably, according to L^2AW, they incur a small latency overhead but ensure high throughput.

4.2 *Lazy-ALRs* for Relaxed-Consistency (RC) and Remote-Asynchronous (RA)

A key class of protocols creates a global order of writes and mandates that all replicas must apply the writes in order. This is called *State Machine Replication* (SMR) [109]. SMR protocols span both the RC and RA classes. We will apply the second variant of ALRs, dubbed *Lazy-ALRs*, to all SMR protocols.

RC SMR protocols, such as ZAB, Derecho, and AllConcur [103], downgrade their consistency guarantees to sequential consistency to read locally under asynchrony. Conversely, in RA protocols such as Raft or Paxos, reads are linearizable but remote and costly, as explained in § 2.4. For instance, in Raft, all follower replicas must send each and every read to the leader, which replies back to them.

Lazy-ALRs. In SMR protocols, all writes across all objects are applied in the same order in all replicas. In other words, a replica can only apply a write if it has applied all preceding writes. Lazy-ALRs leverage this observation by implementing the *sync* as a "fake" write that does not alter any state, but executes the write algorithm. When this "fake" write can be applied locally, then the replica knows that it has also applied all writes preceding the invocation of the reads in its batch, and thus can execute those reads locally. Specifically, *any* replica R executes a batch of Lazy-ALRs as follows:

(1) R forms a batch of read requests and buffers them locally without executing them.
(2) R executes a *sync* using the write algorithm of the protocol. Unlike a normal write, the *sync* does not refer to any object, nor does it carry any new value.
(3) R waits until the *sync* reaches the point where it would be applied locally, if it were a true write. At that time, R locally executes all reads of the batch and responds to the client(s).

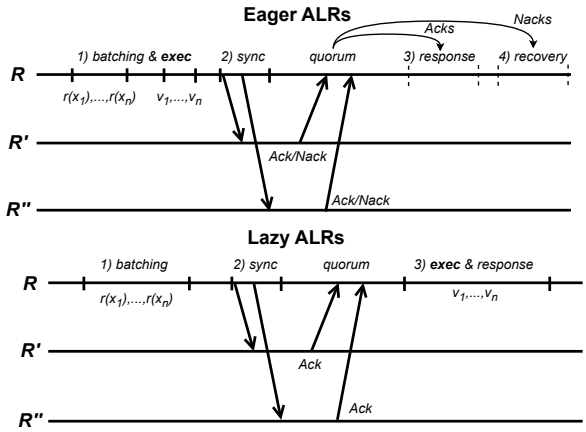

Figure 4: *Eager-ALRs (top) and Lazy-ALRs (bottom) workflow. $r(x_1), ..., r(x_n)$ and $v_1, ..., v_n$ denote the read operations and the values returned, respectively. The key difference is that in Eager-ALRs, reads are executed before the sync, while in Lazy-ALRs, after the sync.*

Correctness. A linearizable read must return a result at least as recent as the last completed write prior the read's invocation. As all writes are applied in the same order to all replicas in SMR. By issuing and completing the *sync* after forming the read batch we ensure that all writes preceding the *sync* in global order are completed and applied locally. Thus, reading after the *sync* completion, guarantees that the Lazy-ALRs return a linearizable (non-stale) value.

Performance. The *sync* has a negligible impact on throughput as 1) its cost is amortized among the reads of a batch and 2) it is a "fake" write, hence has no value to be sent over the network, copied, or applied to storage. Lazy-ALRs allow SMR protocols in the RC class to upgrade their consistency to linearizability, without sacrificing throughput, but at the cost of higher latency for reads. Meanwhile, for SMR protocols in the RA class, Lazy-ALRs can significantly improve throughput. We corroborate these claims in § 6.

Applicability. RA and RC protocols that offer SMR (including ZAB, Raft, Paxos and more) can benefit from Lazy-ALRs.

Summary. Lazy-ALRs add the missing piece for SMR protocols in both RA and RC classes. They allow RA protocols to offer high throughput reads and RC protocols to offer linearizability under asynchrony without sacrificing throughput.

4.3 Performance optimizations

In this section, we brief two optimizations, one for latency and one for throughput, applicable to both ALR variants.

Latency: Opportunistic ALR batching. To amortize the network and compute cost on remote replicas, ALRs work on read batches. Crucially, these batches are formed *opportunistically*: a replica does not wait to form a batch of a specific size; instead, it forms the batch from all currently queued reads. Thus, batching does not affect the latency of a request. If such a batch cannot be formed, this hints that the system is lightly loaded; hence, amortization is not critical. Conversely, when the system is heavily loaded and there is a need for high throughput, there will also be a larger opportunity to form big batches and achieve high amortization.

Throughput: Timely writes for zero-cost sync. In both eager and lazy ALRs, the reading algorithm contains a *sync* operation. If the replica is also about to issue a write, we can leverage the write as a *sync* proxy. Specifically, in the Eager-ALR algorithm, the *sync* queries remote replicas to check whether the reading replica is still in the configuration. If the reading replica happens to also be issuing a write at the same time (i.e., is *timely*), then instead of issuing the *sync* we can simply use the write as a *sync* proxy: if a sufficient quorum of remote replicas acknowledges the write, that implies the replica is still on the most recent configuration.

In Lazy-ALRs the *sync* follows the write algorithm. Similarly, if a write is timely, we can use that as a *sync* proxy. This brings the extra network and computational cost of an ALR-batch on remote replicas to zero. Simply, with timely writes, an ALR-enhanced protocol incurs no extra actions to those executed by the normal writes.

4.4 Real-world use-cases of ALRs

We next present possible real-world use-cases where ALRs can improve throughput, ensure consistency, and tolerate asynchrony.

The RA and RC classes of protocols–such as Raft (e.g., CockroachDB [116], TiKV [57], etcd [2]) and ZAB (e.g., ZooKeeper [58])–are integral to systems demanding strong consistency and high performance. However, RA protocols, typically funnel all reads through a leader, creating a bottleneck. Lazy ALRs address this by enabling follower replicas to serve consistent reads through lightweight syncs, thereby reducing leader load, preserving linearizability, and ensuring that each read incurs minimal or zero network or computation costs to remote replicas. Meanwhile, RC protocols improve performance by allowing local reads but risk stale data under asynchrony; Lazy ALRs remedy these inconsistencies while maintaining as high throughput. Services such as Consul [1], which manages Kubernetes state under heavy autoscaling, and Chubby [27], which handles frequent lock status checks, can benefit from ALR-enabled low-cost load-balanced reads. When Lazy ALRs are applied in RA and RC classes could also benefit large-scale cloud services such as that of Google [16, 35, 114], Meta [5, 89], and Microsoft [21, 62], which mandate on high-throughput on read-dominant workloads and benefit from stronger consistency.

Eager ALRs broaden the application scope of LS protocols—such as Hermes [66]—to asynchronous conditions that frequently encountered in edge or cloud deployments [15, 84]. As these protocols rely on synchrony for correctness, transient delays or congestion can limit their applicability. Eager ALRs address this limitation via a lightweight read validation that has a zero or minimal cost to remote replicas, allowing for high throughput and strong consistency even under asynchrony. Unlike leases, which require tuning to avoid false positives and are unsafe under asynchrony, Eager ALRs ensure safety without sacrificing throughput. This extends the reach of the faster LS protocols e.g., Hermes [47] to real-world environments, reducing reliance on slower protocols—such as Paxos or ABD variants [11, 21, 26, 97, 108].

5 EXPERIMENTAL METHODOLOGY

In this section, we describe the protocols we evaluate and the infrastructure we used to run our experiments.

5.1 Evaluated protocols

We select one protocol from each category and enhance it with ALRs. Specifically, we evaluate:

- Remote-Asynchronous (RA): *Raft* (and *Raft-ALR*)
- Relaxed-Consistency (RC): *ZAB* (and *ZAB-ALR*)
- Local-Synchronous (LS): *Hermes* (and *Hermes-ALR*)

For Hermes, we extend its codebase [65]. For ZAB and Raft, we extend the implementations of the Odyssey framework [47]. We deliberately chose these repositories due to their support of RDMA networking and more importantly, to facilitate the fairest possible comparison, as they provide state-of-the-art protocol implementations that already heavily exploit batching for performance.
We next discuss the three protocols and the respective ALR variants.

Raft (RA). Raft is a leader-based SMR protocol where followers send their reads to the elected leader, who executes them using its local storage. For linearizability, the leader must also collect "heartbeats" from a replica quorum to ensure it is still the leader prior sending the read responses to clients. Raft's leader-only reads are "almost-local" resembling Eager-ALRs. Problematically, Raft limits almost-local reads to the leader node. To get the upper bound performance of Raft, our implementation completes reads locally at the leader (i.e., it omits heartbeats and relaxes linearizability).

> *Raft-ALR:* Despite Raft's leader-only reads resembling Eager-ALRs, we enhance Raft with Lazy-ALRs as it is an SMR protocol. This enables ALRs from *all* replicas. A hybrid of Eager-ALRs and Lazy-ALRs is possible, but we opt solely for Lazy-ALRs for simplicity.

ZAB (RC). ZAB is the leader-based SMR protocol at the heart of Zookeeper [58]. Contrary to Raft, in ZAB, all replicas read locally without any inter-replica communication. This comes at the cost of consistency, as ZAB offers sequential consistency, which is weaker.

> *ZAB-ALR:* As Raft and most crash-tolerant protocols, ZAB is an SMR protocol. We enhance ZAB with Lazy-ALRs, which upgrade its consistency guarantees to linearizability.

Hermes (LS). Hermes is a state-of-the-art crash-tolerant membership-based replication protocol that provides linearizability under (partial-)synchrony, while executing reads locally in all replicas. For its linearizable local reads, each replica holds a time-based lease on the membership and writes must invalidate all lease-holders before completion. However, time-based leases require timing assumptions which, when cease to hold, can lead to linearizability violations.

> *Hermes-ALR:* We replace Hermes' local reads with our Eager-ALRs, removing the time-based leases. This allows Hermes-ALR to offer linearizability under asynchrony.

5.2 Testbed

Infrastructure. To evaluate the performance of ALRs, we conduct an experimental study using R320 servers from Cloudlab [40]. Most of our experiments are conducted on a cluster of 5 servers (replicas) interconnected via a Mellanox switch (SX6036G), while we also evaluate 3 and 7 replicas to study scalability – which are typical for this context [61, 66]. Each machine runs Ubuntu 18.04, has an Intel E5-2630 CPU (8 cores, 2.1Ghz), 16 GB memory, and a double-port 56Gb Mellanox NIC (CX3 PCIe3x8) of which we only use one port. The CPU has 20 MB of L3 cache and two hardware threads per core. We disable turbo-boost and pin threads to cores.

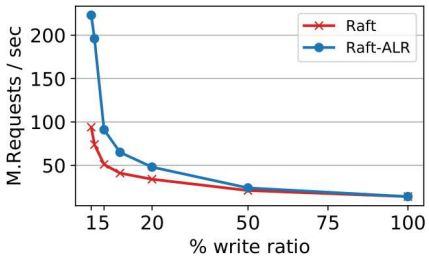

Figure 5: *Raft vs. Raft-ALR throughput. [5 replicas, varying write ratio]*

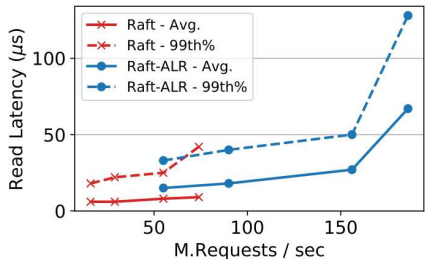

Figure 6: *Raft vs. Raft-ALR latency. [5 replicas, 2% writes, varying load]*

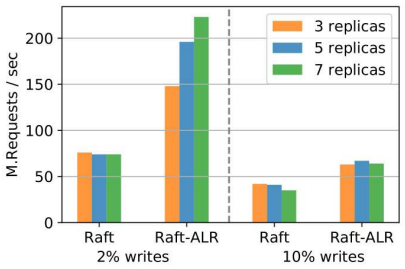

Figure 7: *Raft vs. Raft-ALR scalability. [2% and 10% writes, varying replicas]*

Workload and sensitivity studies. By default, the key-value store (KVS) uses 2MBpages and consists of one million key-value pairs, replicated in all nodes. We use keys and values of 8 and 32 bytes, respectively, which are accessed uniformly. For all evaluated protocols, we use 8 threads (i.e., no hyper-threading) to run the client and protocol logic. The maximum number of reads batched in a single ALR-batch is 128. In § 6.4, we also perform a series of sensitivity studies on larger workload datasets, larger record sizes as well as skewed data accesses based on the YCSB benchmark.

Evaluation fairness. We evaluate the fastest implementations of three state-of-the-art protocols (one per category) and compare them with their ALR-enhanced variants on the same workload and hardware. All three chosen protocol baselines already heavily exploit traditional batching to improve their throughput on reads. Raft and Raft-ALR are evaluated under the exact assumptions (asynchrony and linearizability). We are also not unfair to Hermes or ZAB as our Hermes-ALR and ZAB-ALR are evaluated under strictly more challenging assumptions (never the other way around).

Code availability. The code and the appropriate CloudLab profiles to reproduce our experiments are publicly available. Detailed instructions can be found at: https://law-theorem.com/artifact.pdf.

6 EVALUATION

In § 4, we argued that 1) for RA protocols that cannot read locally, ALRs would significantly increase throughput and 2) for RC and LS protocols which already read locally, ALRs would enhance them with linearizability under asynchrony while resulting in a negligible throughput decrease and just a small latency increase (in the local area network that we target).

We next corroborate our hypotheses by evaluating a representative protocol for each of the classes along with its ALR variant. To do so, we perform a throughput and latency study for each protocol. Another aspect of local reads is that due to their constant overhead, they enable performance scaling with more replicas in read-mostly workloads. We expect that ALRs will maintain this property and perform a scalability study for each protocol to prove it.

Summary of results. As expected, among the three protocols evaluated, only Raft-ALR demonstrates performance gains, achieving significantly higher throughput while preserving linearizability under asynchrony. Both ZAB-ALR and Hermes-ALR maintain similar throughput with a modest latency increase. Notably, unlike their original protocol variants, ZAB-ALR also ensures linearizability, while Hermes-ALR safely operates under asynchrony.

6.1 Remote-Asynchronous (RA): Raft(-ALR)

Raft steers all reads to the leader. Instead, Raft-ALR offers almost-local reads in all replicas. Our hypothesis is that Raft-ALR achieves higher throughput, similar latency and better scalability than Raft.

Throughput while varying write ratio. Figure 5 shows the throughput of Raft and Raft-ALR in million requests per second (M. reqs/s), while varying the writes. Raft-ALR achieves higher throughput than Raft, especially at low write ratios. At 2% writes, Raft-ALR achieves 2.6× higher throughput, 1.8× at 5% writes. This is because in Raft, all reads must be executed by the leader, while in Raft-ALR, all replicas read locally, as long as they wait for a light-weight sync per batch of reads. The benefit of Raft-ALR naturally declines beyond 20% writes, as remote writes at higher write rates involving all replicas dominate the cost. Notably, the throughput benefit comes while also offering linearizability under asynchrony.

Latency while varying load. Figure 6 shows the average and 99th% latency for Raft and Raft-ALR at 2% write ratio, while varying the load. At same load (both at 55 M. reqs/s) the average and tail latencies of Raft (8μs and 25μs) and Raft-ALR (15μs and 33μs) are very close. As the load increases, Raft's leader gets overwhelmed, reaching a tail latency of 47μs at its maximum throughput. At the same load Raft-ALR provides lower tail latency. Specifically, Raft-ALR achieves more than 2× the throughput (156 M. reqs/s) for about the same tail latency, but higher (yet low overall) average latency. This is expected as Raft-ALR is able to load balance its reads across all replicas, alleviating the load on the leader.

Scalability study. Figure 7 shows the throughput of Raft(-ALR) at 2% and 10% writes, when deploying 3, 5, and 7 replicas. This figure validates our hypothesis that almost-local reads allow Raft-ALR to scale its throughput with more replicas better than Raft, which steers all reads to the leader. Naturally, as the replicas increase, the work required to execute a write increases linearly with them, since the write must be applied to all replicas. Conversely, reads should scale gracefully with more replicas. However, Raft cannot scale its throughput with more replicas even at 2% writes. Under more writes (e.g., 10%), the throughput degrades when adding replicas. Conversely, Raft-ALR scales its throughput with more replicas at low write ratios (e.g., 2%). At 10% writes, Raft-ALR slightly increases performance (5 replicas) and keeps the same performance (7 replicas) compared to 3 replicas while offering greater fault tolerance.

Summary. Enhancing Raft with ALRs yields significant throughput gains and better scalability, while achieving similar latency without forfeiting linearizability or asynchrony.

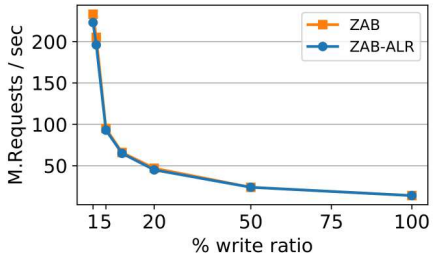

Figure 8: *ZAB vs. ZAB-ALR throughput. [5 replicas, varying write ratio]*

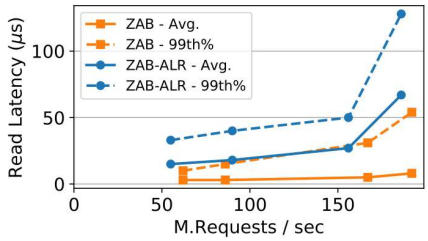

Figure 9: *ZAB vs. ZAB-ALR latency. [5 replicas, 2% writes, varying load]*

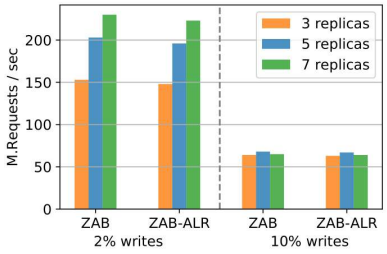

Figure 10: *ZAB vs. ZAB-ALR scalability. [2% and 10% writes, varying replicas]*

6.2 Relaxed-Consistency (RC): ZAB(-ALR)

ZAB executes reads locally but offers weaker consistency. ZAB-ALR offers linearizability under asynchrony via almost-local reads. Our hypothesis is that ZAB-ALR matches ZAB's throughput and scalability, but has higher read latency.

Throughput while varying writes. Figure 8 shows the throughput of ZAB and ZAB-ALR, while varying writes. The throughput of ZAB-ALR is within 4% of ZAB for 1% writes and within 2% at 5% writes. This small gap closes as the writes increase and more read batches in ZAB-ALR exploit timely writes. Because ALRs leverage timely writes, the batch size shrinks with the write ratio. For example, with 10% and 25% writes, 9 and 3 reads are ALR-batched (on average) per write. At 50% writes, both protocols offer the same throughput (24 M. reqs/s). This indicates that the throughput of almost-local reads approaches that of pure local reads.

Latency while varying load. Figure 9 shows the average and 99th% latency for ZAB(-ALR) at 2% write ratio. As expected, ZAB-ALR has higher latencies. This is because the linearizable almost-local reads inevitably include network communication, unlike ZAB's relaxed local reads. Still, ZAB-ALR's average and tail latencies are tight and low. In practice, the average and tail latency are below 27μs and 50μs, respectively, even under high load (e.g., 150 M. reqs/s).

Scalability study. Figure 10 shows the throughput of ZAB and ZAB-ALR at 2% and 10% writes, when scaling replicas. The premise was that almost-local reads afford the same scalability as local reads. Here, ZAB-ALR scales its throughput similarly to ZAB. Specifically, ZAB and ZAB-ALR scale as the number of replicas increase at 2% writes. At 10% writes, both provide a minimal performance improvement when scaling from 3 to 5 replicas and a small performance degradation when scaling to 7 replicas.

Summary. The above experiments showed that ZAB-ALR incurs slightly higher latency compared to ZAB, but matches its throughput and scalability while offering linearizability.

6.3 Local-Synchronous (LS): Hermes(-ALR)

Hermes offers linearizable local reads from all replicas, but under synchrony. Hermes-ALR enables linearizable almost-local reads under asynchrony. Our hypothesis is that Hermes-ALR matches Hermes' throughput and scalability, but has higher latency, due to the round-trip in almost-local reads.

Throughput while varying writes. Figure 11 shows the throughput of Hermes(-ALR), while varying the write ratio. Hermes-ALR comes within 11% of Hermes throughput for 1% writes and within 5% at 5% writes. As the write ratio (and timely writes) increase, the gap closes. At 50% write ratio they both offer the same throughput (39.5 M. reqs/s). This confirms our hypothesis that almost-local reads introduce a minimal throughput cost over pure local reads.

Latency while varying load. Figure 12 shows the average and 99th% latency for Hermes and Hermes-ALR under varying loads (throughput) with a 2% write ratio. As expected, Hermes has the lowest average and tail latencies across all protocols (below 25μs even at maximum load) since Hermes follows a load-balanced design that avoids hotspots, ensuring that no single replica gets overloaded and causes latency spikes. As expected, Hermes-ALR has higher latencies due to the additional steps needed to verify the consistency of the replica configuration after performing local reads. Still, Hermes-ALR maintains low average and tail latencies: 24μs and 53μs for 166 M. reqs/s and below 100μs even under maximum load.

Scalability study. Figure 13 shows the throughput of Hermes and Hermes-ALR at 2% and 10% writes, when scaling the replication degree. Hermes-ALR matches the scalability of Hermes, showing that almost-local reads afford similar throughput scaling to local reads. Specifically, Hermes and Hermes-ALR scale as the number of replicas increases for both 2% and 10% write ratio. As expected, scaling provides higher performance benefit in lower write ratios.

Summary. The above studies demonstrated that Hermes-ALR incurs a small latency cost compared to Hermes but matches its throughput and scalability, while being safe under asynchrony.

6.4 Sensitivity analysis using Hermes(-ALR)

We selected Hermes—the protocol where ALRs exhibit the highest overhead—for a series of sensitivity studies. These studies evaluate performance with a larger dataset (up to 64 million records, the maximum supported by Hermes' underlying datastore) and assess the overhead of ALRs across varying record sizes (32B, 256B, and 1KB). Finally, we analyze the impact of skewed data accesses using Zipfian workload (with exponent a=0.99) modeled after real-world scenarios as defined in the YCSB benchmark [34].

Dataset size. Figure 14 demonstrates that Hermes-ALR incurs low overhead compared to Hermes at 1–10% write ratios, particularly with larger datasets. This suggests that at lower write rates and larger datasets, which result in more memory accesses, the cost of ALRs is further reduced. For write ratios exceeding 20%, the write cost dominates, and the throughput between Hermes-ALR and Hermes matches for both dataset sizes.

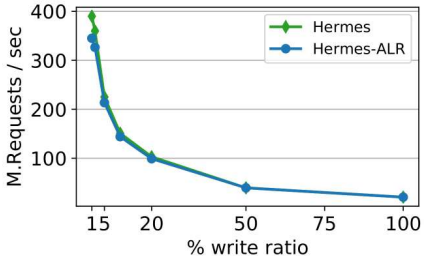

Figure 11: *Hermes vs. Hermes-ALR throughput. [5 replicas, varying writes]*

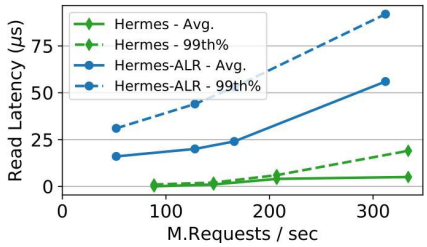

Figure 12: *Hermes vs. Hermes-ALR latency. [5 replicas, 2% writes, varying load]*

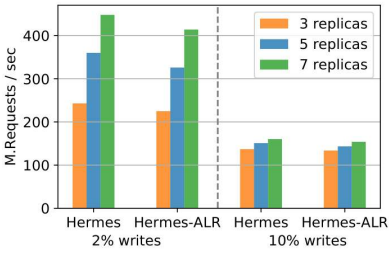

Figure 13: *Hermes vs. Hermes-ALR scalability. [2% and 10% writes, varying replicas]*

Value size. Figure 15 shows that the throughput gap of Hermes and Hermes-ALR is small for smaller record sizes (32B) and becomes even smaller as the record size increases (256B and 1KB). This shows that the relative cost of ALRs diminishes with larger records, as data transfer and processing outweighs the amortized sync cost.

Uniform vs. skewed (YCSB). Figure 16 shows that under both uniform and skewed access patterns (as defined by the YCSB benchmark), the throughput gap between Hermes and Hermes-ALR is small at low write ratios (<10%) and narrows further as the writes increase, closing beyond 10%. This suggests that the coordination overhead of ALRs remains minimal and is not significantly influenced by workloads following real-world skewed accesses. Notably, the throughput under skew is higher than the uniform because Hermes(-ALR) coalesce requests to the popular keys [66].

Summary. The studies show that even for protocols like Hermes, where ALRs exhibit relatively higher overhead, the performance impact remains minimal. Across varying dataset sizes, record sizes, and access patterns, Hermes-ALR maintains similar trends and high throughput, with the gap closing at around 10% write ratios.

7 RELATED WORK

We first focus on practice by elaborating on existing protocols and systems that can be improved via our exposed three-way trade-off and ALRs. Subsequently, we detail existing theoretical results relevant to the L^2AW impossibility before discussing ALRs.

7.1 Practice: Protocols and systems

Some crash-tolerant systems relax one or more L^2AW properties and their design can be improved by applying our ALR techniques to provide linearizability, asynchrony, and almost-local reads. Notably, this work focuses solely on the performance of reads; optimizing writes is out of scope and is left as future work.

Local-Synchronous (LS). Most protocols here use leases to enable local reads without sacrificing linearizability. Leases typically protect either the elected leader [27, 30, 85, 128] or the whole replica configuration [16, 19, 28, 35, 39, 66–68, 95]. The Eager-ALR scheme can be applied to these protocols to eliminate the need for synchrony and ensure safety when time bounds no longer hold.

Alternatively, failure detectors [29], such as those used in Chain Replication [123] and Quema et al.[53], could enable linearizable local reads. However, even weaker failure detectors cannot be implemented in asynchrony[105]. We believe these protocols could replace their fault detectors with the Eager-ALR scheme, possibly combining it with configuration-based algorithms like Vertical Paxos [79], Virtual Synchrony [20], or Alistarh et al.'s proposal [6]. Further exploration of this approach is left for future work.

Relaxed-Consistency (RC). Some crash-tolerant protocols that allow for local reads under asynchrony are not L^2AW-optimal, as they offer weaker consistency – e.g., eventual or read-your-writes [24, 33, 110, 111, 113, 118, 122]. We focus on those providing more intuitive guarantees. Most SMR and atomic broadcast protocols offer sequential consistency while executing local reads [21, 59, 103, 107]. These protocols can apply Lazy-ALRs to ensure linearizability under asynchrony. As shown in our evaluation, this enhancement over ZAB comes at a small cost in latency while delivering almost the same throughput and the strongest consistency.

Remote-Asynchronous (RA). To the best of our knowledge, besides protocols enhanced with our ALR schemes, there are no asynchronous crash-tolerant protocols offering linearizable almost-local reads from *all* replicas. There exist numerous crash-tolerant protocols that offer linearizable but costly remote reads from one or more replicas [9, 11, 20, 26, 31, 36, 43, 69, 70, 75–78, 83, 92–94, 100, 102, 108, 126], thus incurring a significant penalty on read-dominant workloads. Most of these protocols follow the SMR scheme and can use Lazy-ALRs to boost their performance without compromises.

7.2 Theory: Relevant impossibility results

Transactions. Several recent works explored the performance limits of read-only transactions [8, 37, 71, 87, 88, 120]. L^2AW studies the limits of reads over a simple register; thus, it is a stronger impossibility in that respect. Although ALRscould be used to improve read-only transactions, that is outside of the scope of this work.

Atomic registers. Several works involve (a)synchrony and read performance in atomic (linearizable) read/write registers. Attiya & Welch [12] showed that in synchronous, non-crash-tolerant registers, some reads cannot be local, establishing a lower bound for the worst-case read. In this context, coherence protocols offer linearizable local reads, boosting multiprocessor performance [98]. In § 3.3, we discuss other distributed protocols, such as ccKVS [46].

The closest results to ours are that of asynchronous crash-tolerant atomic registers. Dutta et al.[41] showed that it is impossible to construct a crash-tolerant atomic register where both reads and writes complete in one round-trip. Schwarzmann et al.[49] extended this, showing that it's also impossible for each read to complete fast. These results imply that all asynchronous implementations need more than one round-trip for *some* reads to multiple replicas, setting a worst-case bound on reads. However, they don't reveal how fast

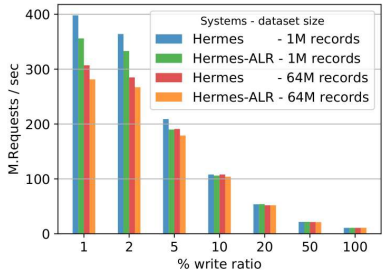

Figure 14: *Hermes vs. Hermes-ALR - small (1M) and large (64M) dataset. [5 replicas]*

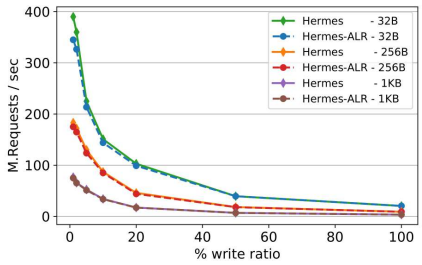

Figure 15: *Hermes vs. Hermes-ALR throughput varying record size. [5 replicas]*

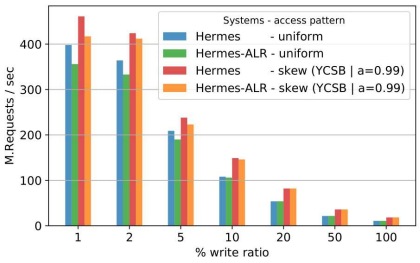

Figure 16: *Hermes vs. Hermes-ALR uniform and YCSB skewed (zipfian a=0.99) accesses.*

any reads can be, nor whether a replica can serve any linearizable read locally. Recently, Schwarzmann et al. [55] emphasized the need for studying efficient implementations with *some* linearizable local reads. The L^{2}AW proves that such implementations are impossible.

FLP. The FLP impossibility [45] concerns the problem of consensus. It asserts that a crash-tolerant asynchronous implementation *cannot always* attain *agreement* (i.e., all nodes must decide the same value) and *termination* (i.e., all non-crashed nodes eventually decide). In short, the FLP presents a tradeoff between safety and liveness on consensus under crashes and asynchrony. The L^{2}AW theorem assumes a similar context, where up to one crash might occur under asynchrony. However, the L^{2}AW focuses on a tradeoff between the safety and performance (i.e., read locality) of a read/write register.

CAP. As in L^{2}AW, the CAP theorem [23, 50] identifies a three-way tradeoff for read/write registers under asynchrony, stating that no implementation can always achieve all three: (1) linearizability, (2) availability, and (3) partition tolerance. In partitioned networks, systems must sacrifice either linearizability or read/write progress.

The CAP does not address read locality. In its crash-free context, protocols like a crash-free variant of Raft can offer linearizable local reads and writes during partition-free execution and even continue serving local reads in a sub-partition if the leader remains connected. In contrast, L^{2}AW proves that under asynchrony, no crash-tolerant protocol can provide a single linearizable local read—even without partitions or crashes—highlighting a stricter impossibility than CAP, which only forfeits safety or progress during network partitions.

Abadi [3] extended CAP informally, suggesting a consistency-latency tradeoff without fault tolerance. Yet, systems exist that support linearizable local reads in non-fault-tolerant settings (§ 3.3). L^{2}AW better captures this tradeoff, showing that crash-tolerant algorithms cannot achieve linearizable local reads under asynchrony.

7.3 Related work and discussion on ALRs

ALRs vs. traditional batching. Traditional batching improves read throughput, albeit imposes network and remote replica compute cost linear to the batch size. The key idea of ALR-batching is that each batch's network and remote replica compute cost is small (or zero – with timely writes) and independent of the reads batched. Thus, ALRs nearly match the throughput and scalability of local reads. Note that our evaluation shows ALRs offering benefits on top of protocols that already heavily exploit traditional batching.

ALRs vs. alternatives to async. linearizability. *Syncs* in ALRs may resemble write-back rounds, which ensure linearizable reads in atomic register protocols like ABD [12, 91]. Contrary to ALRs, write-backs incur substantial network and processing costs on every replica for each read – even if traditional batching is applied.

In contrast to ZAB, where each client must issue an individual empty write to ensure its subsequent read is linearizable, a process that degrades programmability and performance, ALR-batches consolidate reads from all clients. Moreover, ALRs piggyback those batches with local (timely) writes to achieve linearizability at zero extra network or remote replica computation costs, and without requiring programmer intervention. Our evaluation shows that ALRs nearly match the throughput and scalability of ZAB's local, sequentially consistent reads without requiring empty writes. ALRs also apply to other protocols (e.g., lease-based protocols like Hermes).

Coordination avoidance. Several works in distributed transactional systems target to reduce coordination among nodes (e.g., via locks, leaders, or leases) while preserving transactional guarantees [13, 14, 35, 60, 64, 81, 101, 119, 121, 125, 127]. These works either relax consistency, or asynchrony, or fail to perform (almost-)local reads. None of these works offers linearizable (almost-)local reads under asynchrony. In contrast, ALRs minimize the coordination cost via almost-local reads without compromising consistency or asynchrony (but do not support transactions).

(Non-)applicability of ALRs. ALRs apply to leased-based and SMR protocols, as explained in § 4. However, they are not applicable to (decentralized majority-based non-SMR) protocols like ABD. We have explored lowering the read cost in such protocols, but we could not make it zero-cost or batch-size-independent as in ALRs.

8 CONCLUSION

We demonstrated the fundamental tradeoff between the consistency, asynchrony, and performance of crash-tolerant protocols. Guided by this result, we introduced almost-local reads which enable linearizable, asynchronous reads that are nearly as performant as local reads. We evaluated eager and lazy schemes of ALRs on a broad range of crash-tolerant protocols, demonstrating high performance without compromising asynchrony or consistency. We hope that our result will guide future system and protocol designers and that ALRs will help accelerate existing and new replicated systems while delivering strong consistency under asynchrony.

ACKNOWLEDGMENTS

Among others, we thank A. Miller, M. Kleppmann, and A. Petrov for their feedback. This work is supported by the Greek Ministry of Education & Religious Affairs via the HARSH project (project no. ΥΠ3ΤΑ-0560901) that is carried out within the National Recovery-Resilience Plan "Greece 2.0" with funding from NextGenerationEU.

REFERENCES

[1] Consul configuration store. https://www.consul.io/. (Accessed on 09/06/2025).

[2] etcd Key-Value storage system. https://etcd.io/. (Accessed on 09/06/2025).

[3] Daniel Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, 45(2):37–42, feb 2012.

[4] Marcos K. Aguilera and Michael Walfish. No time for asynchrony. In *Proceedings of the 12th Conference on Hot Topics in Operating Systems*, HotOS'09, page 3, USA, 2009. USENIX Association.

[5] Phillipe Ajoux, Nathan Bronson, Sanjeev Kumar, Wyatt Lloyd, and Kaushik Veeraraghavan. Challenges to Adopting Stronger Consistency at Scale. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, pages 13–13, Berkeley, CA, USA, 2015. USENIX Association.

[6] Dan Alistarh, Seth Gilbert, Rachid Guerraoui, and Corentin Travers. Generating Fast Indulgent Algorithms. *Theory of Computing Systems*, 51(4):404–424, 2012.

[7] Peter Alsberg and John Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering*, ICSE '76, pages 562–570, USA, 1976. IEEE.

[8] Karolos Antoniadis, Diego Didona, Rachid Guerraoui, and Willy Zwaenepoel. The impossibility of fast transactions. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1143–1154, 2020.

[9] Vaibhav Arora, Tanuj Mittal, Divyakant Agrawal, Amr El Abbadi, Xun Xue, Zhiyanan Zhiyanan, and Zhujianfeng Zhujianfeng. Leader or majority: Why have one when you can have both? improving read scalability in raft-like consensus protocols. In *Proceedings of the 9th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'17, page 14, USA, 2017. USENIX Association.

[10] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-scale Key-value Store. *SIGMETRICS Perform. Eval. Rev.*, 40(1):53–64, June 2012.

[11] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing Memory Robustly in Message-passing Systems. *J. ACM*, 42(1):124–142, 1995.

[12] Hagit Attiya and Jennifer Welch. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 12(2):91–122, May 1994.

[13] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination avoidance in database systems. *Proc. VLDB Endow.*, 8(3):185–196, November 2014.

[14] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Scalable atomic visibility with ramp transactions. *ACM Trans. Database Syst.*, 41(3), July 2016.

[15] Peter Bailis and Kyle Kingsbury. The network is reliable: An informal survey of real-world communications failures. *Queue*, 12(7):20–32, jul 2014.

[16] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, Asilomar, CA, 2011. .

[17] Luis Andre Barroso, Jimmy Clidaras, and Urs Hoelzle. *The Datacenter as a Computer:An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool, 2013.

[18] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, 2017.

[19] Changyu Bi, Vassos Hadzilacos, and Sam Toueg. Parameterized algorithm for replicated objects with local reads. 2022.

[20] Ken Birman and Thomas Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, pages 123–138, USA, 1987. ACM.

[21] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. Paxos Replicated State Machines As the Basis of a High-performance Data Store. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 141–154, USA, 2011. USENIX Association.

[22] Eric Brewer. Towards robust distributed systems. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 7–, USA, 2000. ACM.

[23] Eric Brewer. CAP twelve years later: How the" rules" have changed. *Computer*, 45(2):23–29, 2012.

[24] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 Conference on Annual Technical Conference*, ATC'13, pages 49–60, Berkeley, 2013. USENIX.

[25] Sebastian Burckhardt. Principles of Eventual Consistency. *Found. Trends Program. Lang.*, 1(1-2):1–150, October 2014.

[26] Matthew Burke, Audrey Cheng, and Wyatt Lloyd. Gryff: Unifying consensus and shared registers. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 591–617, Santa Clara, CA, February 2020. USENIX Association.

[27] Mike Burrows. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 24–24, USA, 2006. USENIX Association.

[28] Tushar Chandra, Vassos Hadzilacos, and Sam Toueg. An algorithm for replicated objects with efficient reads. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC '16, pages 325–334, New York, NY, USA, 2016. ACM.

[29] Tushar Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[30] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos Made Live: An Engineering Perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 398–407, USA, 2007. ACM.

[31] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. Linearizable quorum reads in paxos. In *Proceedings of the 11th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'19, page 8, USA, 2019. USENIX Association.

[32] Audrey Cheng, Xiao Shi, Aaron Kabcenell, Shilpa Lawande, Hamza Qadeer, Jason Chan, Harrison Tin, Ryan Zhao, Peter Bailis, Mahesh Balakrishnan, Nathan Bronson, Natacha Crooks, and Ion Stoica. Taobench: An end-to-end benchmark for social network workloads. *Proc. VLDB Endow.*, 15(9):1965–1977, may 2022.

[33] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, aug 2008.

[34] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[35] James Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):22, 2013.

[36] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. P4xos: Consensus as a network service. *IEEE/ACM Trans. Netw.*, 28(4):1726–1738, aug 2020.

[37] Diego Didona, Panagiota Fatourou, Rachid Guerraoui, Jingjing Wang, and Willy Zwaenepoel. Distributed transactional systems cannot be fast. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '19, page 369–380, New York, NY, USA, 2019. Association for Computing Machinery.

[38] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, 2014. USENIX Association.

[39] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP '15, pages 54–70, New York, 2015. ACM.

[40] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference*, ATC '19, pages 1–14, July 2019.

[41] Partha Dutta, Rachid Guerraoui, Ron R. Levy, and Arindam Chakraborty. How fast can a distributed atomic read be? In *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, page 236–245, New York, NY, USA, 2004. Association for Computing Machinery.

[42] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988.

[43] Niklas Ekström and Seif Haridi. A Fault-Tolerant Sequentially Consistent DSM With a Compositional Correctness Proof, 2016.

[44] A. Fiat and A. Shamir. Generalized 'write-once' memories. *IEEE Trans. Inf. Theor.*, 30(3):470–480, September 2006.

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

[46] Vasilis Gavrielatos, Antonios Katsarakis, Arpit Joshi, Nicolai Oswald, Boris Grot, and Vijay Nagarajan. Scale-out ccNUMA: Exploiting Skew with Strongly Consistent Caching. In *Proceedings of the EuroSys Conference*, EuroSys '18, pages 21:1–21:15, USA, 2018. ACM.

[47] Vasilis Gavrielatos, Antonios Katsarakis, and Vijay Nagarajan. Odyssey: The impact of modern hardware on strongly-consistent replication protocols. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 245–260, New York, NY, USA, 2021. Association for Computing Machinery.

[48] Vasilis Gavrielatos, Antonios Katsarakis, Vijay Nagarajan, Boris Grot, and Arpit Joshi. Kite: Efficient and available release consistency for the datacenter. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '20, page 1–16, New York, NY, USA, 2020. Association for Computing Machinery.

[49] Chryssis Georgiou, Nicolas Nicolaou, and Alexander A. Shvartsman. On the robustness of (semi) fast quorum-based implementations of atomic shared memory. In *Proceedings of the Twenty-Seventh ACM Symposium on Principles of Distributed Computing*, PODC '08, page 425, New York, NY, USA, 2008. Association for Computing Machinery.

[50] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News*, 33(2):51–59, 2002.

[51] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, pages 202–210, New York, NY, USA, 1989. ACM.

[52] Rachid Guerraoui. Indulgent algorithms (preliminary version). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, page 289–297, New York, NY, USA, 2000. Association for Computing Machinery.

[53] Rachid Guerraoui, Dejan Kostic, Ron R. Levy, and Vivien Quema. A High Throughput Atomic Storage Algorithm. In *Proceedings of the 27th International Conference on Distributed Computing Systems*, ICDCS '07, pages 19–, Washington, DC, USA, 2007. IEEE Computer Society.

[54] Theophanis Hadjistasi. Memory access efficiency in distributed atomic object implementations. 2019.

[55] Theophanis Hadjistasi and Alexander A Schwarzmann. Consistent distributed memory services: Resilience and efficiency. In *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[56] Maurice Herlihy and Jeannette Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[57] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. Tidb: a raft-based htap database. *Proceedings of the VLDB Endowment*, 13(12):3072–3084, 2020.

[58] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the USENIX Annual Technical Conference*, USENIX ATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[59] Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, Edward Tremel, Robbert Van Renesse, Sydney Zink, and Kenneth P. Birman. Derecho: Fast state machine replication for cloud services. *Trans. Comput. Syst.*, 36(2):4:1–4:49, 2019.

[60] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-Free Sub-RTT coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 35–49, Renton, WA, April 2018. USENIX Association.

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

[62] Gopal Kakivaya, Lu Xun, Richard Hasha, Shegufta Bakht Ahsan, Todd Pfleiger, Rishi Sinha, Anurag Gupta, Mihail Tarta, Mark Fussell, Vipul Modi, Mansoor Mohsin, Ray Kong, Anmol Ahuja, Oana Platon, Alex Wun, Matthew Snider, Chacko Daniel, Dan Mastrian, Yang Li, Aprameya Rao, Vaishnav Kidambi, Randy Wang, Abhishek Ram, Sumukh Shivaprakash, Rajeet Nair, Alan Warwick, Bharat S. Narasimman, Meng Lin, Jeffrey Chen, Abhay Balkrishna Mhatre, Preetha Subbarayalu, Mert Coskun, and Indranil Gupta. Service Fabric: A Distributed Platform for Building Microservices in the Cloud. In *Proceedings of the EuroSys Conference*, EuroSys '18, pages 1–15, USA, 2018. ACM.

[63] Anuj Kalia, Michael Kaminsky, and David Andersen. Design Guidelines for High Performance RDMA Systems. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, pages 437–450, Berkeley, CA, USA, 2016. USENIX Association.

[64] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, scalable and simple distributed transactions with Two-Sided (RDMA) datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 185–201, Savannah, GA, November 2016. USENIX Association.

[65] Antonios Katsarakis. Hermes protocol. https://github.com/ease-lab/Hermes, 2020. (Accessed on 09/06/2025).

[66] Antonios Katsarakis, Vasilis Gavrielatos, M.R. Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. Hermes: A fast, fault-tolerant and linearizable replication protocol. ASPLOS '20, page 201–217, New York, NY, USA, 2020. Association for Computing Machinery.

[67] Antonios Katsarakis, Yijun Ma, Zhaowei Tan, Andrew Bainbridge, Matthew Balkwill, Aleksandar Dragojevic, Boris Grot, Bozidar Radunovic, and Yongguang Zhang. Zeus: Locality-aware distributed transactions. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 145–161, New York, NY, USA, 2021. Association for Computing Machinery.

[68] Bettina Kemme and Gustavo Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *VLDB*, pages 134–143. Citeseer, 2000.

[69] Gyuyeong Kim and Wonjun Lee. In-network leaderless replication for distributed data stores. *Proc. VLDB Endow.*, 15(7):1337–1349, 2022.

[70] Marios Kogias and Edouard Bugnion. Hovercraft: Achieving scalability and fault-tolerance for microsecond-scale datacenter services. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[71] Kishori M. Konwar, Wyatt Lloyd, Haonan Lu, and Nancy Lynch. Snow revisited: Understanding when ideal read transactions are possible. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 922–931, 2021.

[72] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4):360–391, nov 1992.

[73] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[74] Leslie Lamport. On interprocess communication. *Distributed computing*, 1(2):86–101, 1986.

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

[76] Leslie Lamport. Generalized consensus and Paxos, 2005.

[77] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

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

[79] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the Symposium on Principles of Distributed Computing*, PODC '09, pages 312–313, USA, 2009. ACM.

[80] Kfir Lev-Ari, Edward Bortnikov, Idit Keidar, and Alexander Shraer. Modular Composition of Coordination Services. In *USENIX Annual Technical Conference*, 2016.

[81] Cheng Li, Joao Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. Automating the choice of consistency levels in replicated systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 281–292, Philadelphia, PA, June 2014. USENIX Association.

[82] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, Berkeley, CA, USA, 2012. USENIX Association.

[83] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just Say No to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 467–483, USA, 2016. USENIX Association.

[84] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, page 1–14, New York, NY, USA, 2014. Association for Computing Machinery.

[85] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the harp file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, page 226–238, New York, NY, USA, 1991. Association for Computing Machinery.

[86] Wyatt Lloyd, Michael Freedman, Michael Kaminsky, and David Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the 23rd Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, USA, 2011. ACM.

[87] Haonan Lu, Christopher Hodsdon, Khiem Ngo, Shuai Mu, and Wyatt Lloyd. The snow theorem and latency-optimal read-only transactions. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 135–150, USA, 2016. USENIX Association.

[88] Haonan Lu, Siddhartha Sen, and Wyatt Lloyd. Performance-optimal read-only transactions. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[89] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. Existential consistency: Measuring and understanding consistency at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 295–310, New York, NY, USA, 2015. Association for Computing Machinery.

[90] Aldelir Fernando Luiz, Lau Cheuk Lung, and Miguel Correia. Mitra: byzantine fault-tolerant middleware for transaction processing on replicated databases. *SIGMOD Rec.*, 43(1):32–38, May 2014.

[91] Nancy Lynch and Alexander Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts, 1997.

[92] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: Building Efficient Replicated State Machines for WANs. In *Proceedings of the 8th Conference on Operating Systems Design and Implementation*, OSDI'08, pages 369–384, Berkeley, CA, USA, 2008. USENIX.

[93] Parisa Jalili Marandi, Marco Primi, and Fernando Pedone. High performance state-machine replication. In *Proceedings of the 41st International Conference on Dependable Systems&Networks*, DSN '11, pages 454–465, USA, 2011. IEEE Computer Society.

[94] Iulian Moraru, David Andersen, and Michael Kaminsky. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the 24th Symposium on Operating Systems Principles*, SOSP '13, pages 358–372, USA, 2013. ACM.

[95] Iulian Moraru, David Andersen, and Michael Kaminsky. Paxos quorum leases: Fast reads without sacrificing writes. In *Proceedings of the Symposium on Cloud Computing*, SOCC '14, pages 1–13, USA, 2014. ACM.

[96] Achour Mostéfaoui, Michel Raynal, and Matthieu Roy. Time-efficient read/write register in crash-prone asynchronous message-passing systems. *Computing*, 101(1):3–17, January 2019.

[97] Antoine Murat, Clément Burgelin, Athanasios Xygkis, Igor Zablotchi, Marcos Kawazoe Aguilera, and Rachid Guerraoui. Swarm: Replicating shared disaggregated-memory data in no time. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 24–45, New York, NY, USA, 2024. Association for Computing Machinery.

[98] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, David A. Wood, and Natalie Enright Jerger. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 2nd edition, 2020.

[99] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 385–398, Berkeley, CA, USA, 2013. USENIX Association.

[100] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, USA, 2014. USENIX.

[101] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[102] Marius Poke and Torsten Hoefler. DARE: High-Performance State Machine Replication on RDMA Networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 107–118, USA, 2015. ACM.

[103] Marius Poke, Torsten Hoefler, and Colin W. Glass. AllConcur: Leaderless Concurrent Atomic Broadcast. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '17, pages 205–218, USA, 2017. ACM.

[104] Xiaodong Qi, Zhihao Chen, Zhao Zhang, Cheqing Jin, Aoying Zhou, Haizhen Zhuo, and Quangqing Xu. A byzantine fault tolerant storage for permissioned blockchain. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 2770–2774, New York, NY, USA, 2021. Association for Computing Machinery.

[105] Michel Raynal. Eventual Leader Service in Unreliable Asynchronous Systems: Why? How? Research Report PI 1847, 2007.

[106] Michel Raynal. *Concurrent programming: algorithms, principles, and foundations*. Springer Science & Business Media, 2012.

[107] Benjamin Reed and Flavio P. Junqueira. A Simple Totally Ordered Broadcast Protocol. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, LADIS '08, pages 2:1–2:6, USA, 2008. ACM.

[108] Yasushi Saito, Svend Frølund, Alistair Veitch, Arif Merchant, and Susan Spence. Fab: Building distributed enterprise disk arrays from commodity components. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, page 48–58, New York, NY, USA, 2004. Association for Computing Machinery.

[109] Fred B Schneider. The fail-stop processor approach. *Concurency control and reliability in distributed systems, Chapitre*, 13:370–394, 1987.

[110] William Schultz, Tess Avitabile, and Alyson Cabral. Tunable consistency in mongodb. *Proc. VLDB Endow.*, 12(12):2071–2081, 2019.

[111] Russell Sears, Mark Callaghan, and Eric Brewer. Rose: Compressed, log-structured replication. *Proceedings of the VLDB Endowment*, 1(1):526–537, 2008.

[112] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, page 386–400, Berlin, Heidelberg, 2011. Springer-Verlag.

[113] Xiao Shi, Scott Pruett, Kevin Doherty, Jinyu Han, Dmitri Petrov, Jim Carrig, John Hugg, and Nathan Bronson. *FlightTracker: Consistency across Read-Optimized Online Stores at Facebook*. USENIX Association, USA, 2020.

[114] Jeff Shute, Mircea Oancea, Stephan Ellner, Ben Handy, Eric Rollins, Bart Samwel, Radek Vingralek, Chad Whipkey, Xin Chen, Beat Jegerlehner, Kyle Littlefield, and Phoenix Tong. F1: the fault-tolerant distributed rdbms supporting google's ad business. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, page 777–778, New York, NY, USA, 2012. Association for Computing Machinery.

[115] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. F1: A distributed sql database that scales. *Proc. VLDB Endow.*, 6(11):1068–1079, aug 2013.

[116] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1493–1509, New York, NY, USA, 2020. Association for Computing Machinery.

[117] Jeff Terrace and Michael J. Freedman. Object Storage on CRAQ: High-throughput Chain Replication for Read-mostly Workloads. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, pages 11–11, Berkeley, CA, USA, 2009. USENIX Association.

[118] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 309–324, New York, NY, USA, 2013. Association for Computing Machinery.

[119] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, page 1–12, New York, NY, USA, 2012. Association for Computing Machinery.

[120] Alejandro Z. Tomsic, Manuel Bravo, and Marc Shapiro. Distributed transactional reads: The strong, the quick, the fresh & the impossible. In *Proceedings of the 19th International Middleware Conference*, Middleware '18, page 120–133, New York, NY, USA, 2018. Association for Computing Machinery.

[121] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 18–32, New York, NY, USA, 2013. Association for Computing Machinery.

[122] Albert van der Linde, João Leitão, and Nuno M. Preguiça. Practical client-side replication: Weak consistency semantics for insecure settings. *Proc. VLDB Endow.*, 13(11):2590–2605, 2020.

[123] Robbert van Renesse and Fred B. Schneider. Chain Replication for Supporting High Throughput and Availability. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*, OSDI'04, pages 7–7, Berkeley, CA, USA, 2004. USENIX.

[124] Werner Vogels. Eventually Consistent. *Commun. ACM*, 52(1):40–44, 2009.

[125] Chenggang Wu, Jose M. Faleiro, Yihan Lin, and Joseph M. Hellerstein. Anna: A kvs for any scale. *IEEE Transactions on Knowledge and Data Engineering*, 33(2):344–358, 2021.

[126] HamidReza Zare, Viveck R. Cadambe, Bhuvan Urgaonkar, Nader Alfares, Praneet Soni, Chetan Sharma, and Arif Merchant. Legostore: A linearizable geo-distributed store combining replication and erasure coding. *Proc. VLDB Endow.*, 15(10):2201–2215, 2022.

[127] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. *ACM Trans. Comput. Syst.*, 35(4):12:1–12:37, December 2018.

[128] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan R. K. Ports, Ion Stoica, and Xin Jin. Harmonia: Near-linear scalability for replicated storage with in-network conflict detection. *Proc. VLDB Endow.*, 13(3):376–389, November 2019.