Hermes: a Fast, Fault-Tolerant and Linearizable Replication Protocol

Hermes: a Fast, Fault-Tolerant and Linearizable Replication Protocol

Antonios Katsarakis, Vasilis Gavrielatos, M. R. Siavash Katebzadeh,
Arpit Joshi*, Aleksandar Dragojevic†, Boris Grot, Vijay Nagarajan
University of Edinburgh, *Intel, †Microsoft Research

Abstract

Today's datacenter applications are underpinned by datastores that are responsible for providing availability, consistency, and performance. For high availability in the presence of failures, these datastores replicate data across several nodes. This is accomplished with the help of a *reliable replication protocol* that is responsible for maintaining the replicas strongly-consistent even when faults occur. Strong consistency is preferred to weaker consistency models that cannot guarantee an intuitive behavior for the clients. Furthermore, to accommodate high demand at real-time latencies, datastores must deliver high throughput and low latency.

This work introduces Hermes[1], a broadcast-based reliable replication protocol for in-memory datastores that provides both high throughput and low latency by enabling local reads and fully-concurrent fast writes at all replicas. Hermes couples *logical timestamps* with cache-coherence-inspired *invalidations* to guarantee linearizability, avoid write serialization at a centralized ordering point, resolve write conflicts locally at each replica (hence ensuring that writes never abort) and provide fault-tolerance via replayable writes. Our implementation of Hermes over an RDMA-enabled reliable datastore with five replicas shows that Hermes consistently achieves higher throughput than state-of-the-art RDMA-based reliable protocols (ZAB and CRAQ) across all write ratios while also significantly reducing tail latency. At 5% writes, the tail latency of Hermes is 3.6× lower than that of CRAQ and ZAB.

CCS Concepts • **Computer systems organization** → *Cloud computing*; *Reliability*; *Availability*; • **Software and its engineering** → *Consistency*;

Keywords Fault-tolerant; Replication; Consistency; Availability; Throughput; Latency; Linearizability; RDMA

ACM Reference format:
Antonios Katsarakis, Vasilis Gavrielatos, M. R. Siavash Katebzadeh, Arpit Joshi*, Aleksandar Dragojevic†, Boris Grot, Vijay Nagarajan. 2020. Hermes: a Fast, Fault-Tolerant and Linearizable Replication Protocol. In *Proceedings of Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16–20, 2020 (ASPLOS '20),* 17 pages.
https://doi.org/10.1145/3373376.3378496

[1]The name is inspired by the immortal Olympian figure, who was the messenger of the gods and a conductor of souls into the afterlife.

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland
2020. ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378496

1 Introduction

Today's online services and cloud applications rely on high-performance datastores[2], such as key-value stores (KVS) and lock services, for storing and accessing their dataset. These datastores must provide high throughput at very low latencies while offering high availability, as they are deployed on failure-prone commodity infrastructure [27]. Keeping the dataset in-memory and exploiting high-performance datacenter networking (e.g., RDMA) is essential, but not sufficient.

Data replication is a fundamental feature of high performance and reliable datastores. Data must be replicated across multiple nodes to increase throughput because a single node often cannot keep up with the request load [23]. Replication is also necessary to guarantee that a failure of a node or a network link does not render a portion of the dataset inaccessible. Maintaining the replicas strongly-consistent, to ensure that the services running on the datastore operate correctly, is a challenge, especially in the presence of failures. A *reliable replication protocol* is responsible for keeping the replicas of a datastore strongly-consistent – even when faults occur – by determining the necessary actions for the execution of reads and writes.

When it comes to performance, recent works on reliably-replicated datastores focus on throughput [96] and tend to ignore latency. Meanwhile, latency is emerging as a critical design goal in the age of interactive services and machine actors [16]. For instance, Anwar et al. [7] note that a deep learning system running on top of a reliable datastore is profoundly affected by the latency of the datastore.

Today's replication protocols are not designed to handle the latency challenge of in-memory reliable datastores. Chain Replication (CR) [98], a state-of-the-art high performance reliable replication protocol [7] is a striking example of trading latency for throughput. Our detailed study of CRAQ [96], the state-of-the-art CR variant, reveals that whilst CRAQ can offer very high throughput, it is ill-suited for latency-sensitive workloads. CRAQ organizes the replicas in a chain. While reads can be served locally by each of the replicas, writes expose the entire length of the chain. Moreover, when a read

[2]We use the term datastore broadly to encompass a wide range of in-memory storage systems with an API for reading and writing objects (keys).

reads	writes
local	decentralized
load-balanced	inter-key concurrent
	fast (e.g., few RTTs)

Table 1. *Replication protocol features for high-performance*

hits a key for which a write is in progress, the read incurs an additional latency as it waits for the write to be resolved. With high-latency writes, and mixed-latency reads, CRAQ fails to provide predictably low latency.

This work addresses the challenge of designing a reliable replication protocol that provides both high throughput and low latency within a datacenter. To that end, we identify key features necessary for high performance, which are summarized in Table 1. For reads, this means the ability to execute a read locally on any of the replicas. For writes, high performance mandates the ability to execute writes in a decentralized manner (i.e., any replica can initiate and drive a write to completion without serializing it through another node), concurrently execute writes to different keys, and complete writes fast (e.g., by minimizing round-trips).

Based on these insights, we introduce Hermes, a strongly-consistent fault-tolerant replication protocol for in-memory datastores that provides high throughput and low latency. At a high level, Hermes is a broadcast-based protocol for single-key reads, writes and RMWs that resembles two-phase commit (2PC) [40]. However, 2PC is not reliable (§7) and is overkill for replicating single-key writes. In contrast, Hermes is highly optimized for single-key operations and is reliable.

Hermes combines two ideas to achieve high performance. The first is the use of *invalidations*, which is a form of lightweight locking inspired by cache coherence protocols. The second is per-key *logical timestamps* implemented as Lamport clocks [62]. Together, these enable linearizability, local reads and fully-concurrent, decentralized, and fast writes. Logical timestamps further allow each node to locally establish a single global order of writes to a key, which enables conflict-free write resolution (i.e., writes never abort[3] – another difference from 2PC) and *write replays* to handle faults. To summarize, the contributions of this work are as follows:

- **Introduces *Hermes*, a reliable replication protocol** that utilizes invalidations and logical timestamps to achieve high performance and linearizability. Any replica in Hermes allows for efficient local reads and fast fully-concurrent writes. Hermes handles message loss and node failures by guaranteeing that any write can always be safely replayed.
- **Formally verifies *Hermes*** in *TLA*$^+$ [63] for safety and absence of deadlocks in the presence of crash-stop failures, message reorderings and duplicates.
- **Implements a high-performance RDMA-based reliable KVS** incorporating Hermes with *Wings*, our efficient RDMA RPC library. Our evaluation of Hermes shows that it outperforms the state-of-the-art RDMA-optimized virtual Paxos [50] protocol by an order of magnitude. Moreover, Hermes achieves higher throughput than the highly-optimized RDMA-based state-of-the-art ZAB [53] and CRAQ [96] replication protocols across all write ratios while significantly reducing the tail latency. At 5% writes, the tail latency of Hermes is at least 3.6× lower than that of CRAQ and ZAB.

[3]Read-Modify-Writes (RMWs) in Hermes may abort (§3.6).

2 Background

2.1 In-Memory Distributed Datastores

This work focuses on a replication protocol that can be deployed over datastores, replicated within a local area network such as a datacenter. Clients typically interact with a datastore by first establishing a session through which they issue read and write requests. These datastores keep the application dataset in-memory and employ efficient communication primitives (e.g., RDMA or DPDK) to achieve high throughput at very low latencies. One example of such datastores is key-value stores (KVS) [23, 31, 32, 60, 70] that serve as the backbone for many of today's data-intensive online services, including e-commerce and social networks. Another example is lock services, such as Apache Zookeeper [48] and Google's Chubby [24], which provide an API to the clients that allows them to maintain critical data, including locks.

2.2 Replication and Consistency

Datastores typically partition the stored data into smaller pieces called *shards* and replicate each shard to guarantee fault tolerance. A fault-tolerant replication protocol is then deployed to enforce consistency and fault tolerance across all replicas of a given shard. The number of replicas for a shard is the *replication degree*, and it presents a trade-off between cost and fault tolerance: more replicas increase fault tolerance, but also increase the cost of the deployment. A replication degree between 3 to 7 replicas is commonly considered to offer a good balance between safety and cost [48]. Thus, although a datastore may span numerous nodes, the replication protocol need only scale with the replication degree.

Whenever data are replicated, a consistency model must be enforced. While weak consistency can be leveraged to increase performance, it can also lead to nasty surprises when developers or clients attempt to reason about the system's behavior [100]. For this reason, this work focuses on *reliable replication protocols* that offer the strongest consistency model: *Linearizability* (Lin) [46], which mandates that each request appears to take effect globally and instantaneously at some point between its invocation and completion. Lin has intuitive behavior, is compositional, and allows for the broadest spectrum of applications [45, 99].

2.3 High Performance

Maintaining high performance under strong consistency and fault tolerance is an established challenge [11, 98]. In the context of in-memory datastores, high performance is accepted

to mean low latency and high throughput. Requirements for achieving high performance differ for reads and writes.

Reads The key to achieving both low latency and high throughput on reads is (1) being able to service a read on any replica, which we call *load-balanced reads*, and (2) completing the read locally (i.e., without engaging other replicas). While seemingly trivial, load-balanced local reads (referred to as just *local reads* from now on) are a challenge for many reliable protocols, which may require communication among nodes to agree on a read value (e.g., ABD [9, 74] and Paxos [64]) or that mandate that only a single replica serve linearizable reads for a given key (e.g., Primary-backup [5]).

Writes Achieving high write performance under strong consistency and fault tolerance is notoriously difficult. We identify the following requirements necessary for low-latency high-throughput writes:

➢ *Decentralized*: In order to reduce network hops and preserve load balance across the replica ensemble, any replica must be able to initiate a write and drive it to completion (by communicating with the rest of the replicas) whilst avoiding centralized serialization points. For instance, both ZAB and CR require all writes to initiate at a particular node, hence failing to achieve decentralized writes.

➢ *Inter-key concurrent*: Independent writes on different keys should be able to proceed in parallel, to enable intra- and multi-threaded parallel request execution. For example, ZAB requires all writes to be serialized through a leader, thus failing to provide inter-key concurrency.

➢ *Fast*: Fast writes require minimizing the number of message round-trips, avoiding long message chains (e.g., in contrast to CR), and shunning techniques that otherwise increase write latency (e.g., performing writes in lock-step [75, 88]).

2.4 Reliable Replication Protocols

Failure model We consider a partially synchronous system [34] where processes are equipped with loosely synchronized clocks (LSCs)[4] and crash-stop or network failures may occur (as in [25]). In this model, processes may fail by crashing and their operation is non-Byzantine. Additionally, network failures can manifest as either (1) message reordering, duplication and loss, or (2) link failures that may lead to network partitions.

Reliable replication protocols capable of dealing with failures under the above failure model can be classified into two categories: *majority-based* protocols, which are typically variants of Paxos [64], and protocols that require a stable membership of live nodes (*membership-based* protocols).

Majority-based protocols This class of protocols requires the majority of nodes to respond in order to commit a write, making it naturally tolerant to failures provided that a majority is responsive. However, majority-based protocols pay the price in performance since – in the absence of responses from all replicas – there is no guarantee that a given write has reached all replicas, which makes linearizable local reads fundamentally challenging. Thus, most majority-based protocols give up on local reads but may support decentralized or inter-key concurrent writes [64, 74, 78]. Majority-based protocols that allow for local reads either serialize independent writes on a master (e.g., ZAB) or require communication-intensive per-key leases (§7); problematically, both approaches hurt performance even in the absence of faults.

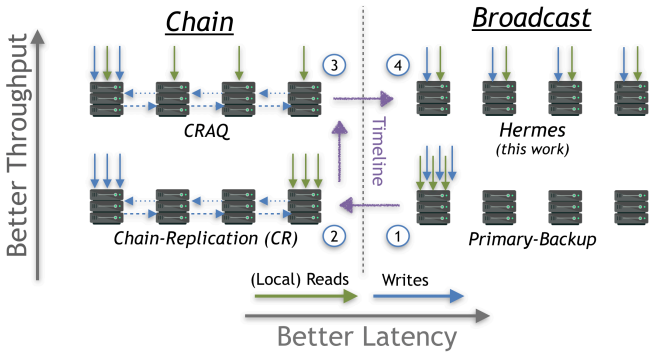


Figure 1. *Comparison of reliable membership-based protocols in terms of throughput and latency.*

Membership-based protocols Protocols in this class require *all operational* nodes in the replica group to acknowledge each write (i.e., read-one/write-all protocols [51]). In doing so, they assure that a committed write has reached all replicas in the ensemble, which naturally facilitates local reads without necessarily hindering write performance. Thus, in the absence of faults, membership-based protocols are naturally free of performance limitations associated with majority-based protocols.

Membership-based protocols are supported by a *reliable membership* (RM) [54, 93], typically based on Vertical Paxos [68]. Vertical Paxos uses a majority-based protocol to reliably maintain a stable membership of *live* nodes [97] (i.e., as in virtual synchrony [18]), which is guarded by leases. Informally, nodes in Vertical Paxos locally store a lease, a membership variable and an *epoch_id*. Nodes are *operational* as long as their lease is valid. Messages are tagged with the epoch_id of the sender at the time of message creation, and a receiver drops any message tagged with a different epoch_id than its local epoch_id. The membership variable establishes the set of live nodes, which allows for efficient execution of reads and writes on any node with a valid lease. During failure-free operation, membership leases are regularly renewed. When a failure is suspected, the membership variable is updated reliably (and epoch_id is incremented) through a majority-based protocol but only after the expiration of leases. This circumvents potential false-positives of unreliable failure detection and maintains safety under network partitions (§3.4). Simply put, updating the membership variable only

[4] Some reliable replication protocols can maintain safety and liveness without LSCs. We discuss one such variant of our Hermes protocol in §8.

after lease expiration ensures that unresponsive nodes have stopped serving requests before they are removed from the membership and new requests complete only amongst the remaining live nodes of the updated membership group.

A common practice for high-performance replication is to optimize for the typical failure-free operation by harnessing the performance benefits of membership-based protocols and limiting the usage of majority-based protocols to RM reconfiguration [33, 51, 68]. In fact, major datacenter operators, such as Microsoft, not only exploit membership-based protocols in their datastores [33, 93], but they also provide LSCs [29, 89] and RM [54] as datacenter services to ease the deployment of membership-based protocols by third parties.

One of the earliest membership-based protocols is Primary-backup [5], which serves all requests at a primary node and does not leverage the backup replicas for performance. Chain Replication (CR) [98] improves upon Primary-backup by organizing the nodes in a chain and dividing the responsibilities of the primary amongst the *head* and the *tail* of the chain, as shown in Figure 1 (bottom-left). CR is a common choice for implementing high performance reliable replication [7, 12, 52, 96, 102]. We next discuss CRAQ [96], a highly optimized variant of CR.

2.5 CRAQ

CRAQ is a state-of-the-art membership-based protocol that offers high throughput and strong consistency (Lin). In CRAQ, nodes are organized in a chain and writes are directed to its head, as in CR. The head propagates the write down the chain, which completes once it reaches the tail. Subsequently, the tail propagates acknowledgment messages upstream towards the head, letting all nodes know about the write's completion.

CRAQ improves upon CR by enabling read requests to be served locally from all nodes, as shown in Figure 1 (top-left). However, if a non-tail node is attempting to serve a read for which it has seen a write message propagating downstream from head to tail, but has not seen the acknowledgement propagating up, then the tail must be queried to find out whether the write has been applied or not.

CRAQ is the state-of-the-art reliable replication protocol that achieves high throughput via a combination of local reads and inter-key concurrent writes. However, CRAQ fails to satisfy the low latency requirement: while reads are typically local and thus very fast, writes must traverse multiple nodes sequentially incurring a prohibitive latency overhead.

3 Hermes

Hermes is a reliable membership-based broadcasting protocol that offers high throughput and low latency whilst providing linearizable reads, writes, and RMWs (single-key transactions). Hermes optimizes for the common case of no failures [15] and targets intra-datacenter in-memory datastores with a replication degree typical of today's deployments (3-7 replicas) [48]. As noted in §2.2, the replica count does not constrain the size of a sharded datastore, since each shard is replicated independently of other shards. Example applications that can benefit from Hermes include reliable datastores [11, 12, 80, 102], lock-services [24, 48] and applications that require high performance, strong consistency and availability (e.g., [1, 20, 103]).

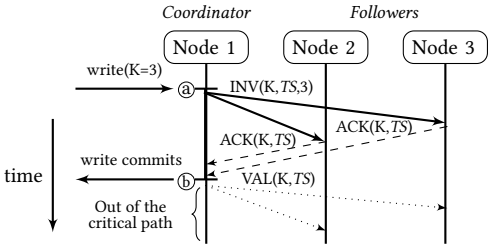


Figure 2. *Example of writing a value of 3 to key K. Nodes one, two and three hold a replica of K.* TS *is the timestamp.*

3.1 Overview

In Hermes, reads complete locally. Writes can be initiated by any replica and complete fast regardless of conflicts. As illustrated in Figure 2, a write to a key proceeds as follows: the replica initiating the write (called *coordinator*) broadcasts an *Invalidation (INV)* message to the rest of the replicas (called *followers*) and waits on *acknowledgments (ACKs)*. Once all ACKs have been received; the write completes via a *Validation (VAL)* message broadcast by the coordinator replica.

We now briefly overview the salient features of Hermes and discuss the specifics in the following subsections.

Invalidations When an INV message is received, the target key is placed in an Invalid state, meaning that reads to the key cannot be served. While conceptually similar to a lock (e.g., in 2PC), the key difference is that with invalidations, concurrent writes to the same key do not fail and are resolved in place through the use of logical timestamps as discussed below. The use of invalidations is inspired by cache coherence protocols, where a cache line in an Invalid state informs the readers that they must wait for an updated value.

Logical timestamps Each write in Hermes is tagged with a monotonically-increasing per-key logical timestamp, implemented using Lamport clocks [62] and computed locally at the coordinator replica. The timestamp is a lexicographically ordered tuple of $[v, c_{id}]$ combining a key's version number (v), which is incremented on every write, with the node id of the coordinator (c_{id}). Two or more writes to a key are *concurrent* if their execution is initiated by different replicas holding the same timestamp. Non-concurrent writes to a key are ordered based on their timestamp version, while concurrent writes from different coordinators (same version) are ordered via their c_{id}[5]. Uniquely tagged writes allow each node to locally establish a global order of writes to a key.

[5] More precisely, a timestamp A: $[v_A, c_{idA}]$ is higher than a timestamp B: $[v_B, c_{idB}]$, if either $v_A > v_B$ or $v_A = v_B$ and $c_{idA} > c_{idB}$.

High-performance non-conflicting writes Hermes allows for high-performance writes (§2.3) by maximizing concurrency while maintaining low latency. First, writes in Hermes are executed from any replica in a decentralized manner, eschewing the use of a serialization point (e.g., a leader); thus reducing the number of network hops and ensuring load balance. In contrast to approaches that globally order independent writes for strong consistency (e.g., ZAB – §5.1.1), Hermes allows writes to different keys to proceed in parallel, hence achieving inter-key concurrency. This is accomplished via Hermes' approach of invalidating all operational replicas to achieve linearizability. When combined with the logical timestamps, invalidations permit concurrent writes to the same key to be correctly linearized at the endpoints; thus, writes do not appear to conflict, making aborts unnecessary.

Finally, in the absence of a failure, writes in Hermes cost one and a half round-trips (INV→ACK→VAL); however, the exposed latency is just a single round-trip for each node. From the perspective of the coordinator, once all ACKs are received, it is safe to respond to a client because at this point, the write is guaranteed to be visible to all live replicas, and any future read cannot return the old value (i.e., the write is *committed* – Figure 2ⓑ). The followers also observe only a single round-trip (further optimized in §3.3), which starts once an INV arrives; at that point, each follower responds with an ACK and completes the write when a VAL is received.

Safely replayable writes Node and network faults during a write to a key may leave the key in a permanently Invalid state in some or all of the nodes. To prevent this, Hermes allows any invalidated operational replica to replay the write to completion without violating linearizability. This is accomplished using two mechanisms. First, the new value for a key is propagated to the replicas in INV messages (Figure 2ⓐ). Such *early value propagation* guarantees that every invalidated node is aware of the new value. Secondly, logical timestamps enable a precise global ordering of writes in each of the replicas. By combining these ideas, a node that finds a key in an Invalid state for an extended period can safely replay a write by taking on a coordinator role and retransmitting INV messages to the replica ensemble with the *original* timestamp (i.e., original version number and c_{id}), hence preserving the global write order.

The above features afford the following properties:
> *Strong consistency*: By invalidating all replicas of a key at the start of a write, Hermes ensures that a key in a Valid state is guaranteed to hold the most up-to-date value. Hermes enforces the invariant that a read may complete if and only if the key is in a Valid state, which provides linearizability.
> *High performance*: Local reads in concert with high performance broadcast-based non-conflicting writes from any replica help ensure both low latency and high throughput.
> *Fault tolerance*: Hermes uses safely replayable writes to tolerate a range of faults, including message loss, node failures, and network partitions. As a membership-based protocol, Hermes is aided by RM to provide a stable group membership of live nodes in the face of failures and network partitions.

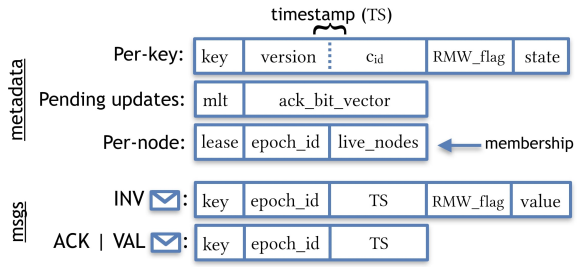


Figure 3. *Metadata stored and messages sent by Hermes.*

3.2 Hermes Protocol in Detail

Hermes protocol consists of four stable states *Valid*, *Invalid*, *Write* and *Replay* and a single transient state *Trans*. Figure 3 illustrates the format of protocol messages and the metadata stored at each replica. A detailed protocol transition table, as well as the *TLA*$^+$ specification, are available online[6].

The following protocol is slightly simplified in that it only focuses on reads and writes (omits RMWs) and only deals with node failures (but not network faults). Resilience to network faults and RMWs are described in §3.4 and §3.6, respectively.

Reads: A read request is serviced on an *operational* replica (i.e., one with an RM lease) by returning the local value of the requested key if it is in the Valid state. If the key is in any other state, the request is stalled.

Writes:

Coordinator

A coordinator node issues a write to a key only if it is in the Valid state; otherwise the write is stalled. To issue and complete a write, the coordinator node:

- **C$_{TS}$**: Updates the key's local timestamp by incrementing its `version` and appending its node id as the c_{id}, and assigns this new timestamp to the write.
- **C$_{INV}$**: Promptly broadcasts an INV message consisting of the `key`, the new timestamp (TS) and the `value` to all replicas and transitions the key to the Write state, whilst applying the new value locally.
- **C$_{ACK}$**: Once the coordinator receives ACKs from all the *live* replicas, the write is completed by transitioning the key to the Valid state (Invalid state if the key was in Trans state[7]).
- **C$_{VAL}$**: Finally, the coordinator broadcasts a VAL consisting of the key and the same timestamp to all the followers.

Note that the coordinator waits for ACKs only from the live replicas as indicated in the membership variable. If a follower fails after an INV has been sent, the coordinator waits

[6] https://hermes-protocol.com

[7] The Trans state indicates a coordinator with a pending write that got invalidated. While not required, the Trans state is useful for tracking when the coordinator's original write completes, hence allowing the coordinator to notify the client of the write's completion.

for the ACK from the failed node until the membership is reliably updated (after the node is detected as failed and the membership lease expires – §2.4). Once the coordinator is not missing any more ACKs, it can safely continue the write.

Follower

- $\mathbf{F_{INV}}$: Upon receiving an INV message, a follower compares the timestamp from the incoming message to its local timestamp of the key. If the received timestamp is higher than the local timestamp, the follower transitions the key to the Invalid state (Trans state if the key was in the Write or the Replay state) and updates the key's local timestamp (both its `version` and c_{id}) and value.
- $\mathbf{F_{ACK}}$: Irrespective of the result of the timestamp comparison, a follower always responds with an ACK containing the same timestamp as that in the INV message of the write.
- $\mathbf{F_{VAL}}$: When a follower receives a VAL message, it transitions the key to the Valid state if and only if the received timestamp is equal to the key's local timestamp. Otherwise, the VAL message is simply ignored.

Write Replays: A request that finds a key in the Invalid state for an extended period of time (determined via the *mlt* timer, described in §3.4) triggers a write replay. The node servicing the request takes on the coordinator role, transitions the key to the Replay state and begins a write replay by re-executing steps $\mathbf{C_{INV}}$ through $\mathbf{C_{VAL}}$ using the TS and value received with the INV message. Note that the original TS is used in the replay (i.e., the c_{id} is that of original coordinator) to allow the write to be correctly linearized. Once the replay is completed, the key transitions to the Valid state after which the initial request is serviced.

▷ **Formal verification**: We expressed Hermes in *TLA*$^+$ [63] and model checked the protocol's reads, writes, RMWs and replays for safety and absence of deadlocks in the presence of message reorderings and duplicates, and membership reconfigurations due to crash-stop failures.

3.3 Hermes Protocol Optimizations

[$\mathbf{O_1}$] *Eliminating unnecessary validations* When the coordinator of a write gathers all of its ACKs but discovers a concurrent write to the same key with a higher timestamp (i.e., was in the Trans state), it does not need to broadcast VAL messages ($\mathbf{C_{VAL}}$), thus saving valuable network bandwidth.

[$\mathbf{O_2}$] *Enhancing fairness* Hermes linearizes writes based on their unique timestamps, consisting of a version and a node id. In case of same versions (i.e., concurrent writes), the linearization is resolved based on the node ids, which might raise concerns about fairness. This is easily mitigated by assigning several *virtual node ids* to each physical node. With this scheme, before issuing a write, a node randomly picks one of its assigned virtual node ids to be used for the write's logical timestamp. Of course, to maintain correctness, the same virtual node id cannot be assigned to more than one physical node. For example, given three nodes (*A, B, and C*), the following sets of virtual ids *A*:{1, 4, 7, 10}, *B*:{2, 5, 8, 11}, and *C*:{3, 6, 9, 12} are safe and would increase fairness.

[$\mathbf{O_3}$] *Reducing blocking latency* In the failure-free case, and during a write to a key, followers block reads to that key for up to a round-trip (§3.1). This blocking latency can be reduced to a half round-trip if followers broadcast ACKs to all replicas instead of just responding to the coordinator of the write ($\mathbf{F_{ACK}}$). Once all ACKs have been received by a follower, it can service the reads to that key without waiting for the VAL message. While this optimization increases the number of ACKs, the actual bandwidth cost is minimal as ACK messages have a small constant size. The bandwidth cost is further offset by avoiding the need to broadcast VAL messages. Thus, under the typical small replication degrees, this optimization comes at negligible cost in bandwidth.

3.4 Network Faults, Reconfiguration and Recovery

This section presents Hermes' operation under imperfect links, network partitions and the transient period of membership reconfiguration on a fault. It then provides an overview of the mechanism to add new nodes to the replica group.

Imperfect Links In typical multi-path datacenter networks, messages can be reordered, duplicated, or lost [36, 39, 73]. Hermes operates correctly under all of these scenarios as described below. In Hermes, the information necessary to linearize operations is embedded with the keys and in the messages in the form of logical timestamps. Thus, even if messages get delayed, reordered, or duplicated in the network, the protocol never violates linearizability.

Hermes uses the same idea of replaying writes if any of its INV, ACK, or VAL messages is suspected to be lost. A message is suspected to be lost for a key if the request's *message-loss timeout (mlt)*, within which every write request is expected to be completed, is exceeded. To detect the loss of an INV or ACK for a particular write, the coordinator of the write resets the request's mlt once it broadcasts INV messages. If the mlt of a key is exceeded before its write completion, then the coordinator suspects a potential message loss and resets the request's mlt before retransmitting the write's INV broadcast.

In contrast, the loss of a VAL message is handled by the follower using a write replay. Once a follower receives a request for a key in the Invalid state, it resets the request's message-loss timeout. If the timestamp or the state has not been updated within the mlt duration, it suspects the loss of a VAL message and triggers a write replay. Although a write replay will never compromise the safety of the protocol, we note that a carefully calibrated timeout will reduce unnecessary replays (e.g., when messages are not lost).

Network Partitions Datacenter network topologies are highly redundant [39, 94]; however, in rare cases, link failures might result in a network partition. According to the CAP theorem [21, 38], either consistency or availability must be sacrificed in the presence of network partitions. Hermes

follows the guidelines of Brewer [22] to permit the datastore to continue serving requests only in its *primary partition*, which is a partition with the majority of replicas. Although failure detectors cannot differentiate between node failures and network partitions, the membership can only be reliably updated in the primary partition – due to its majority-based protocol – and does so only after the expiration of the membership leases. Thus, replicas in a minority partition stop serving requests before the membership is updated and new requests are able to complete only in the primary partition. While this approach allows the datastore to continue operating even under network partitions, it reduces Hermes resilience from $n-1$ node failures to tolerating less than $\lfloor \frac{n}{2} \rfloor$ failures, if the RM protocol is run by the datastore replicas and not external nodes. Nevertheless, this cost is similar to any other reliable protocol that tolerates network partitions [48, 50, 64]. Once network connectivity is restored, nodes previously on a minority side can re-join the replica group via a recovery procedure explained below.

Membership reconfiguration after a failure Following a network partition or a node failure and expiration of the leases for all of the nodes in a membership group, a majority-based protocol is used to reliably update the membership. We refer to this update as *m-update*, which consists of a lease renewal, a new list of live nodes and an incremented epoch_id. Although the m-update is consistent even in the presence of faults, the update does not reach all live replicas instantaneously. Rather, there is a transient period when some replicas that are considered live, according to the latest value of the membership, have received the m-update while others have not and are still non-operational.

Hermes seamlessly deals with the transition of m-update without violating safety. Hermes' replicas which have received the m-update are able to act as coordinators and serve new requests. Thus, reads that find the target key in the Valid state can immediately be served as usual. In contrast, writes or reads that require a replay (i.e., targeted key is Invalid) are effectively stalled until all live nodes as indicated by the membership variable receive the m-update. This is because writes and write replays do not commit until all live replicas become operational and acknowledge their INV messages.

During this transition period, any live follower that has not yet received the latest m-update will simply drop the INV messages, because those messages are tagged with an `epoch_id` greater than the follower's local `epoch_id`. This manifests as a simple message loss to a coordinator which triggers retransmission of the INVs (§3.4). The coordinator eventually completes its writes once all live followers have received the latest membership and become operational.

Recovery Hermes' fault tolerance properties enable a datastore to continue operating even in the presence of failures. However, as nodes fail, new nodes need to be added to the datastore to continue operating at peak performance. To add a new node, the membership is reliably updated, following which all other live replicas are notified of the new node's intention to join the replica group. Once all the replicas acknowledge this notification, the new node starts operating as a *shadow replica* that participates as a follower for all of the writes but does not serve any client requests. Additionally, it reads chunks (multiple keys) from other replicas to fetch the latest values and reconstruct the datastore similarly to existing approaches [33, 84]. After reading the entire datastore, the shadow replica is up-to-date and transitions to operational state, whereby it is able to serve client requests.

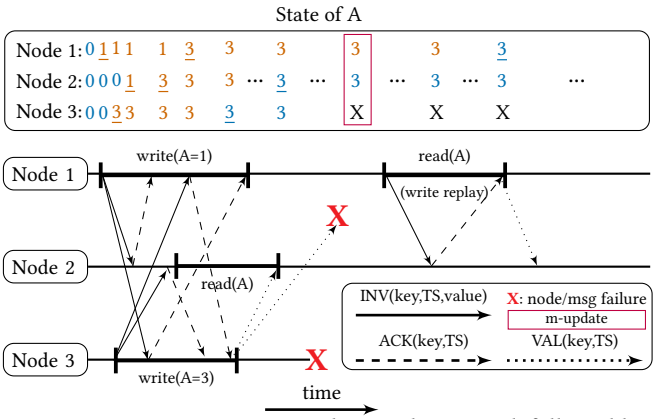


Figure 4. *Concurrent writes to key A, then a read, followed by a node and a message failure which trigger a write replay on the last read. State of A shows the values of the replicas; blue represents Valid state, orange represents other states. Underlined values indicate a change in value and/or state.*

3.5 Operational Example

In this subsection, we discuss Figure 4, which illustrates an example of Hermes' execution with reads and writes to key A. The purpose is to demonstrate the operation of Hermes while shedding light onto some of its corner cases in the presence of concurrency and failures. For simplicity, we assume no use of virtual node ids or any latency optimizations (§3.3).

First, node 1 initiates a write ($A = 1$), by incrementing its local timestamp, broadcasting INV messages (solid lines) and transitioning key A to Write state. Similarly, node 3 initiates another concurrent write ($A = 3$). Recall that INVs in Hermes contain the key, the timestamp (including the c_{id}), and the value to be written. We assume that key A is initially stored with the same value (zero) and timestamp in all three nodes.

Node 2 ACKs the INV message from node 1 (dashed line), updates its timestamp and value, and transitions key A to Invalid state. Node 3 ACKs the INV of node 1, but it does not modify A or its state since its local timestamp is higher (same version but higher c_{id}). Subsequently, node 2 receives the INV from node 3, which has a bigger timestamp than the locally stored timestamp, resulting in an update in its local value and timestamp, all while remaining in Invalid state. Likewise, node 1 ACKs the INV of node 3, by updating the

value, the timestamp, and transitioning to Trans state.

Meanwhile, node 2 starts a read, but it is stalled since its local copy of *A* is invalidated. Once node 3 receives all of the ACKs, it completes its own write by transitioning *A* to the Valid state and broadcasts a VAL message (dotted lines) to the other replicas. When node 2 receives node's 3 VAL message, it transitions *A* to Valid state and completes its stalled read.

Once node 1 receives all of the ACKs it completes its write but transitions to the Invalid state. This occurs because the write from node 3 took precedence over its own due to a higher timestamp, but the VAL from node 3 has not yet been received. Note that although the write from node 1 completes later than the concurrent write from node 3, it is linearized before the write of node 3 due to its lower timestamp (c_{id}).

As a last step, we consider a failure scenario, whereby the VAL message from node 3 to node 1 gets dropped and node 3 crashes. Thus, key *A* in node 1 remains in the Invalid state. Once leases expire and node 3 is detected as failed, the membership variable is reliably updated. Subsequently, node 1 receives a read for *A*, but finds *A* invalidated by a failed node. Thus, node 1 triggers a write replay by broadcasting INV messages with the key's locally-stored timestamp and value (i.e., replaying node 3's original write). Crucially, the fact that INV messages contain both the timestamp and value to be written allows node 1 to safely replay node 3's write. Node 2 ACKs the INV from node 1 without applying it, since it already has the same timestamp. Once node 1 gets the ACK from node 2, it is able to unblock itself. Lastly, node 1 completes the replay of the write by broadcasting a VAL message to all of the live nodes (i.e., node 2, in this example).

3.6 Read-Modify-Writes in Hermes

So far, we have focused on read and write operations; however, Hermes also supports read-modify-write (RMW) atomics that are useful for synchronization (e.g., a compare-and-swap to acquire a lock). In general, atomic execution of a read followed by a write to a key may fail if naively implemented with simple reads and writes. This is because a read followed by a write to a key is not guaranteed to be performed atomically since another concurrent write to the same key with a smaller logical timestamp could be linearized in-between the read-write pair, hence violating the RMW semantics.

For this reason, an RMW update in Hermes is executed similarly to a write, but it is conflicting. Hermes may abort an RMW which is concurrently executed with another *update* operation (either a write or another RMW) to the same key. Hermes commits an RMW if and only if the RMW has the highest timestamp amongst any concurrent updates to that key. Moreover, it purposefully assigns higher timestamps to writes compared to their concurrent RMWs. As a result, any write racing with an RMW to a given key is guaranteed to have a higher timestamp, thus safely aborting the RMW. Meanwhile, if only RMW updates are racing, the RMW with the highest node id will commit, and the rest will abort.

More formally, Hermes always maintains safety and guarantees progress in the absence of faults by ensuring two properties: (1) *writes always commit*, and (2) *at most one of possible concurrent RMWs to a key commits*. To maintain these properties, the following protocol alterations are needed:

- **Metadata**: To distinguish between RMW and write updates, an additional binary flag (`RMW_flag`) is included in INV messages. The flag is also stored in the per-key metadata to accommodate *update replays*.
- $\mathbf{C_{TS}}$: When a coordinator issues an update, the version of the logical timestamp is incremented by one if the update is an RMW and by two if it is a write.
- $\mathbf{F_{RMW\text{-}ACK}}$: A follower ACKs an INV message for an RMW only if its timestamp is equal to or higher than the local one; otherwise, the follower responds with an INV based on its local state (i.e., same message used for write replay).
- $\mathbf{C_{RMW\text{-}abort}}$: In contrast to non-conflicting writes, an RMW with pending ACKs is aborted if its coordinator receives an INV to the same key with a higher timestamp.
- $\mathbf{C_{RMW\text{-}replay}}$: After an RM reconfiguration, the coordinator resets any gathered ACKs of a pending RMW and replays the RMW to ensure it is not conflicting.

3.7 Summary

This section introduced Hermes, a reliable membership-based protocol that guarantees linearizability. Hermes' decentralized broadcast-based design is engineered for high throughput and low latency. By leveraging invalidations and logical timestamps, Hermes enables efficient local reads and high-performance updates that are decentralized, fast, and inter-key concurrent. Writes (but not RMWs) in Hermes are also non-conflicting. Finally, Hermes seamlessly recovers from a range of node and network faults thanks to its write replays, enabled by *early value propagation* and logical timestamps.

4 System

To evaluate the benefits and limitations of the Hermes protocol, we build HermesKV, an in-memory RDMA-based KVS with a typical read/write API. HermesKV is replicated across all the machines comprising a deployment and relies on the Hermes protocol to ensure the consistency of the deployment. We choose RDMA networking to match the trend in modern datacenters towards offloaded network stacks and ultra-low latency fabrics instead of onloaded UDP/TCP [44, 77].

In §4.1, we present a functional overview of the HermesKV and briefly outline the implementation of its KVS. Subsequently, we describe *Wings* (§4.2), our RDMA-based library which serves as the communication layer of the HermesKV.

4.1 Overview and KVS

Each node in HermesKV is composed of a number of identical *worker* threads. Each worker performs the following tasks: 1) decodes client requests; 2) accesses the local KVS

replica; and 3) runs the Hermes protocol to complete requests. Client requests are distributed among the worker threads of the system. Requests can be either reads or writes. Worker threads communicate solely to coordinate writes (and write replays) as reads are completed locally.

Our KVS is based on ccKVS [37], which is a version of MICA [70] (found in [57]), but modified to support CRCW using seqlocks [61]. Seqlocks are beneficial as they allow for efficient lock-free reads [92]. We further extend ccKVS to accommodate the Hermes-specific protocol actions, state transitions and request replies based on the replica state.

The Hermes protocol is agnostic to the choice of a datastore and can be used with any datastore. We choose ccKVS since its minimalist design allows us to focus on the impact of the replication protocol itself without regard of idiosyncrasies or overheads of a commercial-grade datastore.

4.2 Wings: an RDMA RPC layer for Hermes

State-of-the-art RDMA-based KVS designs such as HERD [55] and ccKVS [37] have shown Remote Procedure Calls (RPCs) to be a highly effective design paradigm. Hence, we leverage RDMA Unreliable Datagram sends (UD sends) to build the *Wings* library, a simple and efficient RPC layer over RDMA. Wings allows for opportunistic batching of multiple messages into one network packet, implements application-level flow control, provides support for broadcasts and enlists an array of RDMA low-level optimizations.

Opportunistic Batching The benefits of batching multiple application-level messages into a single network packet are well-known. Batching amortizes the network header overhead, leading to better utilization of network bandwidth.

Wings automatically performs opportunistic batching for all messages. The programmer provides Wings with a buffer that holds messages that need to be sent to various remote servers. Wings inspects the buffer in order to batch messages with the same receiver, then it creates a lightweight application-level header per batch specifying how many messages are batched and sends the packets. Note that the batching performed by Wings is *opportunistic*, as it will never stall in order to form a batch; rather, Wings creates batches for the intended receivers only with readily available messages.

Broadcast Primitive Wings implements software-based broadcasts as a series of unicasts to all members of a broadcast group. Wings performs opportunistic batching for broadcasts in a similar manner as regular requests.

Flow Control Wings uses credit-based flow control [59] to manage the data flow between the servers of a deployment. The programmer can specify whether the credit updates are *explicit* or *implicit*. Implicit credits are common in a communication pattern where a server sends a request and receives a response for that request; the response can be then treated as an implicit credit update. HermesKV leverages this feature when coordinating a write: the coordinator broadcasts invalidations to all remote replicas and treats the acknowledgments as credits updates. Explicit credits are needed for messages that do not require responses. HermesKV exploits explicit credits for the validation messages, as the protocol does not require validations to be acked. Instead, after receiving several validation messages, HermesKV nodes send explicit credit messages to the sender to inform it of their buffer availability. Similarly to other Wings operations, explicit credits are opportunistically batched. The receiver polls a number of incoming messages and sends back a single explicit credit update message.

RDMA Optimizations In Wings, we build RDMA RPCs over UD sends following published low-level guidelines [13, 37, 56, 57]. Transparent to the programmer, Wings amortizes and alleviates PCIe overheads. First, Wings performs doorbell batching and selective signaling when sending work requests to the NIC, and it inlines payloads inside the work requests when the payload is small enough (188B on our NIC) to reduce the required NIC-initiated DMAs per work request. Broadcasts are implemented as a linked list of work requests each with a different destination but all pointing to the same payload. Moreover, explicit credit updates are header-only packets exploiting the *immediate* header field [14]. Thus, they are cheaper to transmit and due to the lack of a payload they reduce PCIe transactions on both sender and receiver sides.

5 Experimental Methodology

5.1 Evaluated Systems

We evaluate Hermes by comparing its performance with a majority-based and membership-based RDMA-enabled baseline protocols. To facilitate a fair protocol comparison, we study all protocols over a common multi-threaded KVS implementation based on HermesKV (as described in §4). All protocols are implemented in C over the RDMA *verbs* API [14]. The evaluated systems are as follows:

- **rZAB** : In-house, multi-threaded, RDMA-enabled ZAB [90].
- **rCRAQ**: In-house, multi-threaded, RDMA-based CRAQ [96].
- **HermesKV** : Implementation of Hermes as in §3 and §4, without the latency optimization (O_3 from §3.3).

Our evaluation mainly focuses on the comparison of HermesKV to rZAB and rCRAQ, since they share the KVS and communication library, which allows us to isolate the effect of the protocol itself on performance. We also compare Hermes to **Derecho** [50] (§6.5), the state-of-the-art RDMA-optimized open-source implementation of membership-based (i.e., virtually synchronous) Paxos. Table 2 below summarizes the read and write features of the evaluated systems.

5.1.1 rZAB

In ZAB protocol, one node is the leader and the rest are followers. A client can issue a write to any node, which in turn propagates the write to the leader. The leader receives writes from all nodes, serializes them and proposes them by

System	Local reads		Writes		
	Leases	Consistency	Concurrency	Latency (RTT)	Dec.
HermesKV	one per RM	Lin	inter-key	1	✓
rCRAQ	one per RM	Lin	inter-key	O(n)	✗
rZAB	none	SC	serializes all	2 †	✗
Derecho	none	SC	serializes all	1 ‡	✓

Table 2. *Comparison of read and write features for the evaluated systems. SC: sequentially consistent; RM: reliable membership; Dec: decentralized; n: number of replicas; † 1 RTT for master's writes; ‡lock-step commit.*

broadcasting atomically to all followers. The followers send back acknowledgements (ACKs) to the leader; on receiving a majority of ACKs for a given write, the leader commits the write locally and broadcasts commits to the followers.

A client's read can be served locally by any node without any communication as long as the last write of that client has been applied in that node. However, local reads in ZAB are sequentially consistent (SC), which is weaker than Lin. Problematically, the fact that ZAB is not Lin leads to a performance issue on writes. This is because, in contrast to the stricter Lin, sequential consistency (SC) is not compositional [10]. As a result, it is not possible to deploy independent instances (e.g., per-key) of SC protocols such as ZAB to increase the concurrency of writes because the composition of those instances would violate SC. If a ZAB client requires linearizable reads, then it can issue a *sync* command prior to the read. A sync is completed similarly to a write, necessarily increasing the read latency. We do not evaluate linearizable reads, to get the upper bound performance of the ZAB protocol.

rZAB optimizations We apply to rZAB all HermesKV optimizations and utilize the RDMA Multicast [14] to tolerate ZAB's asymmetric (i.e. leader-oriented) network traffic pattern. Our highly optimized, RDMA implementation of ZAB outperforms the open-source implementation of Zookeeper (evaluated in [52]) by three orders of magnitude. Of course, Zookeeper is a production system incorporating features beyond the ZAB protocol, such as client tracking and checkpointing to disk. By evaluating a lean and optimized version of just ZAB, we are facilitating a fair protocol comparison.

5.1.2 rCRAQ

CRAQ affords local reads and inter-key concurrent, but not decentralized, writes (§2.5 details the CRAQ protocol). We identify two undesirable properties of CRAQ: 1) writes must traverse multiple hops before completing, adversely affecting the system's latency; and 2) the nodes of the chain are generally not well balanced, in terms of the amount of work performed per-packet potentially affecting the system's throughput. To evaluate how these properties affect performance, we study our own RDMA-enable version of CRAQ (rCRAQ), that enjoys all optimizations available in HermesKV.

5.2 Testbed

We conduct our experiments on a cluster of 7 servers interconnected via a 12-port Infiniband switch (Mellanox MX6012F). Each machine runs Ubuntu 18.04 and is equipped with two 10-core CPUs (Intel Xeon E5-2630v4) with 64 GB of system memory and a single-port 56Gb Infiniband NIC (Mellanox MCX455A-FCAT PCI3 x16). Each CPU has 25 MB of L3 cache and two hardware threads per core. We disable turbo-boost, pin threads to cores and use huge pages (2 MB) for the KVS. The KVS consists of one million key-value pairs, replicated in all nodes. Unless stated otherwise, we use keys and values of 8 and 32 bytes, respectively; which are accessed uniformly.

6 Evaluation

6.1 Throughput on Uniform Traffic

Figure 5a shows the performance of HermesKV, rCRAQ and rZAB while varying the write ratio under uniform traffic.

▹ *Read-only:* For read-only, all three systems exhibit identical behaviour, achieving 985 Million Requests per second (MReqs/s), as all systems perform reads locally from all replicas. To reduce clutter we omit the read-only from the figure.

▹ *HermesKV:* At a 1% write ratio (Figure 5a), HermesKV achieves 770 MReqs/s, outperforming both baselines (12% better than rCRAQ and 4.5× better than rZAB). As the write ratio increases, the throughput of HermesKV gradually drops, reaching 72 MReqs/s on a write-only workload. The throughput degradation at higher write ratios is expected because writes require an exchange of messages over the network, which cost both CPU cycles and network bandwidth.

At 20% write ratio, HermesKV significantly outperforms the baselines (40% over rCRAQ, 3.4× over rZAB). The reason for HermesKV's good performance compared to alternatives is that it combines local reads with high-performance writes.

▹ *rCRAQ:* The CRAQ protocol is well-suited for high throughput, comprising both inter-key concurrent writes and local reads. Nevertheless, rCRAQ performs worse than HermesKV across all write ratios, with the gap widening as write ratios increase. That difference has its root in the design of CRAQ.

Firstly, reads in CRAQ are not always local: if a non-tail node is attempting to serve a read for a key for which it has seen a write but not an ACK, then the tail must be queried to find out whether the write has been applied or not. Therefore, increasing the write ratio has an adverse effect on the reads, as more reads need to be served remotely via the tail node.

This disadvantage hints to a more important design flaw: the CRAQ design is heterogeneous, mandating that nodes assume one of three different roles – head, tail or intermediate – where each role has different responsibilities. As such, load is not equally balanced, so the system is always bottlenecked by the node with the heaviest responsibilities. For instance, at high ratios, the tail node is heavily loaded as it receives read queries from all nodes. Meanwhile, at low write ratios, the tail has fewer responsibilities than an intermediate node, as it only propagates acknowledgements up the chain, whilst an intermediate must also propagate writes downstream.

(a) *Uniform* **(b)** *Skewed [Zipfian exponent 0.99]*

Figure 5. *Throughput for* 1% *to* 100% *write ratio. [5 nodes]*

▹ *rZAB:* As expected, ZAB fails to achieve high throughput at non-zero write ratios as it imposes a strict ordering constraint on *all* writes at the leader. The strict ordering makes it difficult to extract concurrency, inevitably causing queuing of writes and delaying the subsequent reads within each session. At 1% write ratio, rZAB achieves 172 MReqs/s, which drops to a mere 16 MReqs/s for a write-only workload.

6.2 Throughput under Skew

We next explore how the evaluated protocols perform under access skew. We study an access pattern that follows a power-law distribution with a Zipfian exponent of 0.99, as in YCSB [28] and recent studies [32, 37, 81]. Figure 5b shows the performance of the three protocols when varying the write ratio from 1% to 100%. We discuss read-only separately.

▹ *Read-only:* Similarly to the uniform read-only setting, all three protocols achieve identical performance (4183 MReq/s) due to their all-local accesses. Unsurprisingly, the read-only performance under the skewed workload is higher than the uniform performance for all protocols. This is because under a skewed workload there is temporal locality among the popular objects, which is captured by the hardware caches.

▹ *HermesKV:* HermesKV gracefully tolerates skewed access patterns, especially at low write ratios (achieving 1190 MReq/s at 1% write ratio). Repeatedly accessing popular objects cannot adversely affect HermesKV write throughput, as concurrent writes to the popular objects can proceed without stalling (as explained in § 3.1). Meanwhile, read throughput thrives under a skewed workload as reads are always local in HermesKV, and as such can benefit from temporal locality.

▹ *rCRAQ:* Similarly, rCRAQ benefits from temporal locality when accessing the local KVS, while write throughput is unaffected by the skew, as multiple writes for the same key can concurrently flow through the chain. The problem, however, is that non-tail nodes cannot complete reads locally if they have seen a write for the same key but have not yet received an ACK. In that case, the tail must be queried. Under skew, such cases become frequent, with reads to popular objects often serviced by the tail and not locally. Thus, at higher write ratios, the tail limits rCRAQ's performance.

▹ *rZAB:* rZAB is not affected by the conflicts created by the skewed access pattern, as it already serializes all writes irrespective of the object they write. In practice, rZAB performs slightly better under skew as hardware caches are more effective due to better temporal locality for popular objects.

6.3 Latency Analysis

6.3.1 Latency vs Throughput

Figure 6a illustrates the median (50th%) and the tail (99th%) latencies of the three protocols as a function of their throughput at 5% write ratio. We measure latency of each request from the beginning of its execution to its completion.

All three systems execute reads locally, while writes incur protocol actions that include traversing the network. Therefore, at 5% write ratio, we expect the median latency of all protocols to be close to the latency of a read and the tail latency to be that of a write. Consequently, the gap between the median and the tail latency is to be expected for all systems and should not be interpreted as unpredictability.

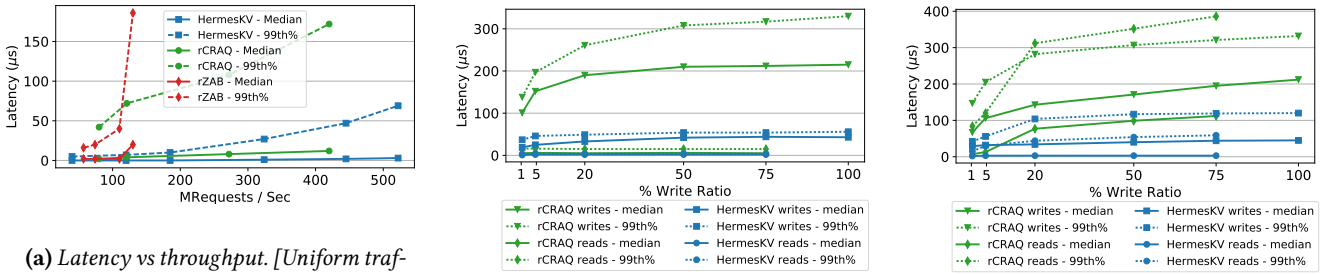
▹ *HermesKV:* The median latency of HermesKV is the latency of a read, and as expected, is consistently very low (on the order of 1μs) even at peak throughput because reads are local. The tail latency is determined by the writes. The tail latency increases with the load, because writes traverse the network and thus can be subject to queuing delays as load increases. At peak throughput, the tail latency of HermesKV is 69μs.

▹ *rCRAQ:* In rCRAQ the median latency is the latency of a read, and as such, is typically on the order of a few microseconds. As expected, the tail latency, which corresponds to a write, is consistently high – at least 3.6× larger than HermesKV at the same throughput points – ranging from 42μs at lowest load to 172μs at peak load. The high write latency is directly attributed to the protocol design as writes in rCRAQ need to traverse multiple network hops, incurring both the inherent network latency and the queuing delays in all the nodes.

▹ *rZAB:* As the other two protocols, rZAB achieves a low median latency because of its local reads, but even at moderate throughput, its tail latency is much larger (e.g., more than 3.6× than that of Hermes at 75MReq/s) because of the high latency of the writes that must serialize on the leader.

6.3.2 Latency vs Write ratio

Figures 6b and 6c depict the median and tail latencies of reads and writes separately, under both skewed and uniform workloads, when operating at peak throughput of CRAQ – which corresponds roughly to 50-85% of HermesKV peak throughput. rZAB cannot achieve high enough throughput



(a) *Latency vs throughput. [Uniform traffic, 5% write ratio]*

(b) *Median and 99th% [Uniform traffic]*

(c) *Median and 99th% [Zipfian 0.99]*

Figure 6. *Latency analysis. [5 nodes]*

to be included in the figures.

▷ *Uniform:* HermesKV delivers very low, tightly distributed latencies across all write ratios, for both reads (2μs-15μs) and writes (29μs-42μs). As expected, rCRAQ exhibits a similar behaviour for reads but not for writes. rCRAQ write latencies are at least 3.9× to 5.9× larger than the corresponding write latencies of HermesKV, with median latencies ranging from 101 to 215μs while the tail latencies range from 138 to 330μs.

▷ *Skew:* Under skew the tail latencies of both reads and writes increase in HermesKV, because reads and writes are more likely to conflict on popular objects. The tail read latency is the latency of a read that stalls waiting for a write to return; not surprisingly that latency is roughly equal to the median latency of a write. Similarly, the tail latency of a HermesKV write increases up to 120μs because in the worst case without failures a write might need to wait an already outstanding write (to the same key) issued from the same node.

In rCRAQ, the latencies of writes remain largely unaffected, compared to the uniform workload. However, the behaviour of reads changes radically because reads are far more likely to conflict with writes under skew; such reads are sent to the tail node. Consequently, the tail node becomes very loaded, which is reflected in both the median (up to 112μs) and tail (up to 386μs) read latencies. This is a very important result; while high write latencies are expected of rCRAQ, we show that reads latencies can suffer as well, making CRAQ an undesirable protocol for systems that target low latency.

6.4 Scalability Study

To investigate the scalability of the evaluated protocols, we measure their performance by varying the replication degree. Figure 7 depicts the throughput of the three protocols under write ratios of 1% and 20% for 3, 5 and 7 machines.

▷ *HermesKV:* Reads in HermesKV are always local and thus their overhead is independent of the number of replicas, allowing HermesKV to take advantage of the added replicas to increase its throughput. Therefore, HermesKV's scalability is dependent on the write ratio, achieving almost linear scalability with the number of replicas at 1% writes, while maintaining its performance advantage at 20% write ratio.

▷ *rCRAQ:* When scaling rCRAQ, the expectations are similar to HermesKV: reads are scalable, but writes are not. However, scaling the replicas in CRAQ implies extending the size of the chain. Consequently, more non-tail nodes redirect their reads to the tail node. Thus, the tail becomes loaded, degrading read throughput, while also creating back-pressure in the chain which adversely affects write throughput. That phenomenon is apparent in Figure 7; at 20% write ratio, rCRAQ throughput degrades when the chain is extended from 5 to 7 nodes.

▷ *rZAB:* rZAB also performs reads locally, and thus is expected to see a benefit from greater degrees of replication at low write ratios. However, write requests incur a large penalty in rZAB, as the leader receives and serializes writes from all machines. When the leader cannot keep up with the write stream, the replicas inevitably fall behind as the reads stall waiting for the writes to complete, and the writes are queued on the leader. Indeed, in Figure 7, we observe that even though rZAB scales well for a read-dominant workload, at a 20% write ratio, increasing the replication degree from 5 to 7 cuts the performance almost in half. Our results are in line with the original scalability analysis of Zookeeper [48].

6.5 Comparison to Derecho

In this section, we compare HermesKV throughput with the RDMA-optimized open-source Derecho [50], the state-of-the-art membership-based variant of Paxos. Derecho's codebase partitions work at each node across several threads (3-4), but does not support higher degrees of threading. For a fairest possible comparison, we limit HermesKV to a single thread.

Figure 8 shows throughput of a write-only workload, while varying the object size from 32B to 1KB – such relatively small object sizes are typical for datastore workloads [8, 70]. Although HermesKV is constrained to a single thread, it outperforms Derecho by an order of magnitude on small object sizes (32B), while maintaining its benefit even on larger objects (3× at 1KB). Derecho increases the performance of its totally ordered writes by exploiting monotonic predicates [50]. Nevertheless, due to its lock-step delivery and its inability to offer inter-key concurrent writes, it fails to match the performance of Hermes. We note that HermesKV's throughput naturally decreases as the object size increases and more bytes per request are transferred.

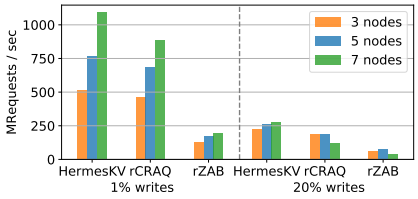


Figure 7. *Scalability study. [Uniform traffic]*

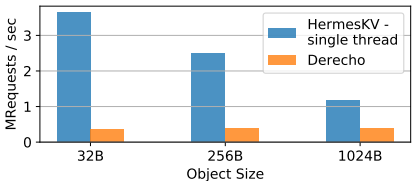


Figure 8. *Comparison to Derecho. [Uniform traffic, 5 nodes, write-only]*

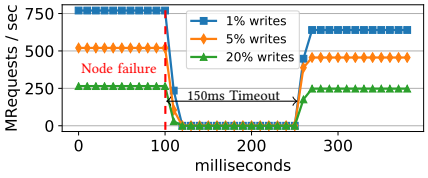


Figure 9. *HermesKV under failure. [Uniform traffic, 5 nodes, timeout=150ms]*

6.6 Throughput with Failures

In order to study the behaviour of HermesKV when a failure occurs, we implement RM in a similar manner with [54] and integrate it with HermesKV. Figure 9 depicts the behaviour of HermesKV when a failure is injected at 1, 5, and 20% write ratios in a five node deployment and a conservative timeout of 150ms. The throughput drops to zero almost immediately after the failure, because all live nodes are blocked waiting for acknowledgements from the failed node. After the timeout expires, the machines reach agreement (via a majority-based protocol) to reliably remove the failed node from the membership, and subsequently continue operating with four nodes. The agreement part of the protocol entails exchanging a handful of small messages over an unloaded RDMA network, which takes just a few microseconds and is not noticeable in the figure. The recovered, steady-state throughput is lower after the failure, because one node is removed from the replica group.

7 Related Work

Consensus and atomic broadcast State machine replication (SMR) [91] provides linearizability by explicitly ordering all client requests (reads and writes), and requiring all replicas to execute the requests in the determined order. SMR can be implemented using any fault-tolerant consensus or atomic broadcast algorithm to order the requests. Numerous such algorithms have been proposed [18, 26, 71, 82], the most popular being variants of Paxos [64]. Recent works present optimized variants of these protocols that exploit commutative operations [2, 65, 78] and rotating coordinators [75]. Others leverage a ring-based topology [6, 42, 85], similarly to CRAQ, to increase throughput but at the cost of latency.

Most of these protocols are majority-based and sacrifice performance for a failure model without RM support. Therefore, they typically enforce strong consistency at the cost of performance, by sacrificing either local reads or concurrency. An abundance of such protocols forfeits local reads [9, 18, 19, 35, 64–67, 69, 75, 76, 78, 83, 87], thus incurring a significant penalty on read-dominant datastore workloads.

Meanwhile, protocols that allow local reads sacrifice performance on writes. A recent atomic broadcast protocol offering local reads does so by relaxing consistency and applying writes in lock-step [88]. Chandra et al. [25], present a protocol with linearizable local reads through object leases, which serializes writes on a leader. ZAB [90], a characteristic example of such protocols, enables local reads and serializes writes on a leader but without using object leases, thus increasing performance but at the cost of consistency. As shown in our evaluation, Hermes significantly outperforms ZAB with its decentralized and inter-key concurrent writes.

Per-key leases Linearizable protocols that use object leases for local reads, such as [11, 25, 79], could be deployed per-key (i.e., one protocol instance for each key) to match the inter-key concurrency, but not latency, of writes in Hermes. However, this mandates a lease for *each* individual key, which is not scalable for realistic datastores with millions of keys. In this approach, for linearizable local reads, leases must be continuously renewed for each key — even in the absence of writes or reads. This renewal costs at least $\Theta(n)$ messages (n = number of replicas) per key and must occur before each lease expires, causing significant network traffic. Moreover, the lease duration cannot be made very long since this would translate into similarly long unavailability upon a fault. In contrast, Hermes, with its invalidating writes and just a single RM lease per replica, offers local reads while being fully inter-key concurrent at a message cost independent of the number of the keys stored by the datastore.

Hardware-assisted replication Some proposals leverage hardware support to reduce the latency of reliable replication, such as FPGA offloading [49] and programmable switches [30, 52, 69]. For instance, Zhu et al. [105] use programmable switches for in-network conflict detection, to allow local reads from any replica. Other works tailor reliable protocols by exploiting RDMA [17, 87, 101]. Hermes offers local reads without hardware support. When evaluated over RDMA, Hermes significantly outperforms Derecho, which represents the state-of-the-art of RDMA-based approaches (§6.5).

Optimized reliable replication A recent work [86] proposed a Primary-backup optimization to reduce the exposed write latency for external clients, but its correctness relies on commutative operations. Howard's optimization [47] allows Paxos to commit after 1 RTT in conflict- and failure-free rounds, albeit reads are not local. In contrast, Hermes is not limited to commutative operations and affords local reads.

Reliable transaction commit Hermes provides single-key linearizabile reads, writes and RMWs, but does not offer fully reliable multi-key transactions. The distributed transaction

commit requires an agreement on whether a transaction should atomically commit or abort: the transaction may only be committed if all parties agree on it. A popular protocol to achieve this is the two-phase commit (2PC) [40]. However, the 2PC is a blocking protocol and must be extended to three phases (3PC) to tolerate coordinator failures [41, 43, 95]. A more common way to achieve reliable transactions is layering a transactional protocol over a reliable replication protocol [29, 58, 104]. For instance, FaRM and Sinfonia use Primary-backup [3, 33]. In this latter setting, Hermes can be used as the underlying reliable replication protocol to increase locality and performance.

Geo-replication Hermes is designed for replication within a local area network (e.g., a datacenter), where network partitions are rare. The conventional wisdom for fault-tolerant replication across dataceneters is to offer causal consistency which allows execution in all sites under partitions. A causal replication protocol could be tiered over several independently-replicated geo-distributed instances managed by Hermes (instead of CR [4, 72]) to accelerate geo-replication.

8 Discussion

Are local reads beneficial in a large-scale datastore? Throughout the paper, we report the latency of operations with respect to a node (replica) in a distributed datastore. In a large-scale datastore, clients might be external and not co-located with a replica they desire to access. Although in this case, reads in Hermes do not provide locality with respect to the client, they still ensure load balance and low latency. This is because, in Hermes, a remote read from an external client would be solely served by just one replica without additional messages, delays or coordination amongst replicas.

Reducing write latency of external clients For the protocols discussed in this work, if clients are external, an additional round-trip is required to reach and get a response from the replica ensemble. Thus, the common-case exposed latency for an external client to commit a write in Hermes is 2 RTTs. To reduce the response time, followers can send ACKs to both the coordinator of the write and the client[8] ($\mathbf{F_{ACK}}$). This reduces the latency to complete linearizable writes from external clients to 1.5 RTTs. The message cost of this optimization (about twice the number of ACKs of the baseline protocol) is linear with the replication degree.

Hermes without Loosely Synchronized Clocks (LSCs) This paper considers a failure model with LSCs. Hermes leverages LSCs only for the RM lease management, to ensure that a node with a lease always has the latest membership. However, Hermes can be efficiently deployed in the absence of LSCs with minor modifications. Hermes' writes seamlessly work without LSCs, since they commit only after all acknowledgments are gathered, which occurs only if the coordinator has the same membership as every other live follower[9].

Linearizable reads in Hermes can also be served without LSCs. The basic idea is to use a committed write to *any* key after the arrival of a read request as a guarantee that the given node is still part of the replica group, hence validating the read. More specifically, observe that a node can establish that it is a member of the latest membership by successfully committing a write. Using this idea, a read at a given node can be speculatively executed but not immediately returned to the client. Once the node executes a subsequent write to any key and receives acknowledgments from a *majority* of replicas, it can be sure that it was part of the latest membership when the read was executed. Once that's established, the read can be safely returned to the client. Note that a majority of acknowledgments suffices because the membership itself is updated via a majority-based protocol.

If a subsequent write is not readily-available (e.g., due to low load) the coordinator can send a *membership-check* message which contains only the membership epoch_id to the followers. The followers will acknowledge this message if they are in the same epoch. After a majority of acknowledgments is collected, the coordinator returns the read. The *membership-check* is a small message and can be issued after a batch of read requests are speculatively executed by the coordinator. Thus, although serving reads without LSCs increases the latency of reads until a majority of replicas respond, it incurs zero (if a subsequent write is timely) or minimal network cost to validate the read.

9 Conclusion

This work introduced Hermes, a membership-based reliable replication protocol that offers both high throughput and low latency. Hermes uses invalidations and logical timestamps to achieve linearizability, with local reads and high-performance updates at all replicas. In the common case of no failures, Hermes broadcast-based writes are non-conflicting and always commit after a single round-trip. Hermes tolerates node and network failures through its safe write replays. An evaluation of Hermes against state-of-the-art protocols shows that it achieves superior throughput at all write ratios and considerably reduces tail latency.

Acknowledgments

We thank our shepherd, Rodrigo Rodrigues, and our anonymous reviewers for their constructive comments and feedback. This work is supported by Microsoft Research and ARM through their PhD Scholarship Programmes, as well as EPSRC grants EP/M027317/1 and EP/L01503X/1.

[8] Coordinator must send an ACK to the client as well.

[9] Followers with different membership value would have otherwise ignored the received INVs due to discrepancy in the message epoch_ids (§2.4)

References

[1] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Larry Kai, Alexander Shraer, Arif Merchant, and Kfir Lev-Ari. 2016. Slicer: Auto-sharding for Datacenter Applications. In *Proceedings of the 12th Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX, USA, 739–753.

[2] Marcos Aguilera, Carole Gallet, Hugues Fauconnier, and Sam Toueg. 2000. Thrifty Generic Broadcast. In *Proceedings of the 14th Conference on Distributed Computing (DISC '00)*. ., London, UK, 268–282.

[3] Marcos Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. 2007. Sinfonia: A New Paradigm for Building Scalable Distributed Systems. *SIGOPS Oper. Syst. Rev.* 41, 6 (2007), 159–174.

[4] Sérgio Almeida, João Leitão, and Luís Rodrigues. 2013. ChainReaction: A Causal+ Consistent Datastore Based on Chain Replication. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*. ACM, New York, NY, USA, 85–98.

[5] Peter Alsberg and John Day. 1976. A Principle for Resilient Sharing of Distributed Resources. In *Proceedings of the 2nd International Conference on Software Engineering (ICSE '76)*. IEEE, USA, 562–570.

[6] Yair Amir, Louise Moser, Peter Melliar, Deborah Agarwal, and Paul Ciarfella. 1995. The Totem Single-ring Ordering and Membership Protocol. *ACM Trans. Comput. Syst.* 13, 4 (Nov. 1995), 311–342.

[7] Ali Anwar, Yue Cheng, Hai Huang, Jingoo Han, Hyogi Sim, Dongyoon Lee, Fred Douglis, and Ali R. Butt. 2018. bespoKV: Application Tailored Scale-out Key-value Stores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*. IEEE Press, Piscataway, NJ, USA, Article 2, 16 pages.

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

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

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

[11] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*. ., Asilomar, CA, 223–234.

[12] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. 2012. CORFU: A Shared Log Design for Flash Clusters. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, USA, 1–1.

[13] Dotan Barak. 2013. Tips and tricks to optimize your RDMA code. https://www.rdmamojo.com/2013/06/08/tips-and-tricks-to-optimize-your-rdma-code/. (June 2013).

[14] Dotan Barak. 2015. RDMA Aware Networks Programming User Manual. (2015).

[15] Luiz Barroso, Urs Hölzle, and Parthasarathy Ranganathan. 2018. The datacenter as a computer: Designing warehouse-scale machines. *Synthesis Lectures on Computer Architecture* 13, 3 (2018), i–189.

[16] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the Killer Microseconds. *Commun. ACM* 60, 4 (2017), 48–54.

[17] Jonathan Behrens, Ken Birman, Sagar Jha, Matthew Milano, Edward Tremel, Eugene Bagdasaryan, Theo Gkountouvas, Weijia Song, and Robbert Van Renesse. 2016. *Derecho: Group Communication at the Speed of Light.* Technical Report. Technical Report. Cornell University.

[18] Ken Birman and Thomas Joseph. 1987. Exploiting Virtual Synchrony in Distributed Systems. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles (SOSP '87)*. ACM, New York, 123–138.

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

[20] Fábio Botelho, Fernando Ramos, Diego Kreutz, and Alysson Bessani. 2013. On the Feasibility of a Consistent and Fault-Tolerant Data Store for SDNs. In *Proceedings of the 2013 Second European Workshop on Software Defined Networks (EWSDN '13)*. IEEE, USA, 38–43.

[21] Eric Brewer. 2000. Towards Robust Distributed Systems. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '00)*. ACM, New York, NY, USA, 7–.

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

[23] 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. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 Conference on Annual Technical Conference (ATC'13)*. USENIX, Berkeley, 49–60.

[24] Mike Burrows. 2006. 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)*. USENIX Association, Berkeley, CA, USA, 24–24.

[25] Tushar Chandra, Vassos Hadzilacos, and Sam Toueg. 2016. An Algorithm for Replicated Objects with Efficient Reads. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing (PODC '16)*. ACM, New York, NY, USA, 325–334.

[26] Tushar Chandra and Sam Toueg. 1996. Unreliable failure detectors for reliable distributed systems. *J. ACM* 43, 2 (1996), 225–267.

[27] Kelly Clay. 2013. Amazon.com Goes Down, Loses $66,240 Per Minute. https://www.forbes.com/sites/kellyclay/2013/08/19/amazon-com-goes-down-loses-66240-per-minute/#4e849f8b495c. (2013).

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

[29] 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. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3 (2013), 22.

[30] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. 2015. NetPaxos: Consensus at Network Speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR '15)*. ACM, New York, Article 5, 7 pages.

[31] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. *SIGOPS Oper. Sys.* 41, 6 (2007), 5–20.

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

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

[34] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus

in the Presence of Partial Synchrony. *J. ACM* 35, 2 (1988), 288–323.

[35] Niklas Ekström and Seif Haridi. 2016. A Fault-Tolerant Sequentially Consistent DSM With a Compositional Correctness Proof. (2016). arXiv:1608.02442

[36] Nathan Farrington. 2009. Multipath TCP under Massive Packet Reordering. (2009).

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

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

[39] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. 2011. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *Proceedings of the ACM SIGCOMM 2011 Conference (SIGCOMM '11)*. ACM, New York, NY, USA, 350–361.

[40] Jim Gray. 1978. Notes on Data Base Operating Systems. In *Operating Systems, An Advanced Course*. Springer-Verlag, London, UK, 393–481.

[41] Rachid Guerraoui. 2002. Non-blocking atomic commit in asynchronous distributed systems with failure detectors. *Distributed Computing* 15, 1 (2002), 17–25.

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

[43] Rachid Guerraoui, Mikel Larrea, and André Schiper. 1995. Non Blocking Atomic Commitment with an Unreliable Failure Detector. In *Proceedings of the 14TH Symposium on Reliable Distributed Systems (SRDS '95)*. IEEE Computer Society, Washington, DC, USA, 41–.

[44] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over Commodity Ethernet at Scale. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. ACM, New York, NY, USA, 202–215.

[45] Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, USA.

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

[47] Heidi Howard. 2019. Distributed consensus revised (Thesis). (2019).

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

[49] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. 2016. Consensus in a Box: Inexpensive Coordination in Hardware. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation (NSDI'16)*. USENIX, Berkeley, CA, USA, 425–438.

[50] Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, Edward Tremel, Robbert Van Renesse, Sydney Zink, and Kenneth P. Birman. 2019. Derecho: Fast State Machine Replication for Cloud Services. *Trans. Comput. Syst.* 36, 2, Article 4 (2019), 49 pages.

[51] Ricardo Jiménez-Peris, M. Patiño Martínez, Gustavo Alonso, and Bettina Kemme. 2003. Are Quorums an Alternative for Data Replication? *ACM Trans. Database Syst.* 28, 3 (Sept. 2003), 257–294.

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

[53] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. 2011. Zab: High-performance Broadcast for Primary-backup Systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks (DSN '11)*. IEEE, Washington, DC, USA, 245–256.

[54] 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. 2018. Service Fabric: A Distributed Platform for Building Microservices in the Cloud. In *Proceedings of the EuroSys Conference (EuroSys '18)*. ACM, USA, 1–15.

[55] Anuj Kalia, Michael Kaminsky, and David Andersen. 2014. Using RDMA Efficiently for Key-value Services. *SIGCOMM Comput. Commun. Rev.* 44, 4 (Aug. 2014), 295–306.

[56] Anuj Kalia, Michael Kaminsky, and David Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-sided (RDMA) Datagram RPCs. In *Proceedings of the 12th Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX, USA, 185–201.

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

[58] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. 2013. MDCC: Multi-data Center Consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*. ACM, New York, NY, USA, 113–126.

[59] H. T. Kung, Trevor Blackwell, and Alan Chapman. 1994. Credit-based Flow Control for ATM Networks: Credit Update Protocol, Adaptive Credit Allocation and Statistical Multiplexing. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications (SIGCOMM '94)*. ACM, New York, NY, USA, 101–114.

[60] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Sys.* 44, 2 (2010), 35–40.

[61] Christoph Lameter. 2005. Effective synchronization on Linux/NUMA systems. (2005).

[62] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565.

[63] Leslie Lamport. 1994. The temporal logic of actions. *Transactions on Programming Languages and Systems (TOPLAS)* 16, 3 (1994), 872–923.

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

[65] Leslie Lamport. 2005. Generalized consensus and Paxos. (2005).

[66] Leslie Lamport. 2006. Fast Paxos. *Distributed Computing* 19, 2 (2006), 79–103.

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

[68] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. 2009. Vertical Paxos and Primary-backup Replication. In *Proceedings of the Symposium on Principles of Distributed Computing (PODC '09)*. ACM, USA, 312–313.

[69] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. 2016. 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)*. USENIX Association, Berkeley, CA, USA, 467–483.

[70] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-memory Key-value Storage. In *Proceedings of the 11th Networked Systems Design and Implementation (NSDI'14)*. USENIX Association, Berkeley, USA, 429–444.

[71] Barbara Liskov and James Cowling. 2012. Viewstamped replication revisited. (2012).

[72] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. 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)*. ACM, USA, 401–416.

[73] Yuanwei Lu, Guo Chen, Bojie Li, Kun Tan, Yongqiang Xiong, Peng

Cheng, Jiansong Zhang, Enhong Chen, and Thomas Moscibroda. 2018. Multi-Path Transport for RDMA in Datacenters. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 357–371.

[74] Nancy Lynch and Alexander Shvartsman. 1997. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. (1997), 272-281 pages.

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

[76] Parisa Jalili Marandi, Marco Primi, and Fernando Pedone. 2011. High Performance State-machine Replication. In *Proceedings of the 41st International Conference on Dependable Systems&Networks (DSN '11)*. IEEE Computer Society, Washington, DC, USA, 454–465.

[77] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. 2019. Snap: A Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. ACM, New York, NY, USA, 399–413.

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

[79] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2014. Paxos Quorum Leases: Fast Reads Without Sacrificing Writes. In *Proceedings of the Symposium on Cloud Computing (SOCC '14)*. ACM, USA, 1–13.

[80] Edmund B. Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. 2012. Flat Datacenter Storage. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX, Hollywood, CA, 1–15.

[81] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2016. The Case for RackOut: Scalable Data Serving Using Rack-Scale Systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC '16)*. ACM, New York, NY, USA, 182–195.

[82] Brian M. Oki and Barbara H. Liskov. 1988. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the Seventh Symposium on Principles of Distributed Computing (PODC '88)*. ACM, New York, NY, USA, 8–17.

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

[84] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. 2011. Fast Crash Recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 29–41.

[85] Parisa Marandi, M. Primi, N. Schiper, and F. Pedone. 2010. Ring Paxos: A high-throughput atomic broadcast protocol. In *2010 International Conference on Dependable Systems Networks*. ., Chicago, USA, 527–536.

[86] Seo Jin Park and John Ousterhout. 2019. Exploiting Commutativity for Practical Fast Replication. In *Proceedings of the 16th Conference on Networked Systems Design and Implementation (NSDI'19)*. USENIX, USA, 47–64.

[87] Marius Poke and Torsten Hoefler. 2015. 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)*. ACM, New York, NY, USA, 107–118.

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

[89] Ian Prittie. 2018. Windows Time Service | Microsoft Docs. https://docs.microsoft.com/en-us/windows-server/networking/windows-time-service/windows-time-service-top. (2018).

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

[91] Fred B. Schneider. 1990. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.* 22, 4 (Dec. 1990), 299–319.

[92] Michael L. Scott. 2013. Shared-Memory Synchronization. (2013).

[93] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojević, Dushyanth Narayanan, and Miguel Castro. 2019. Fast General Distributed Transactions with Opacity. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. ACM, New York, NY, USA, 433–448.

[94] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*. ACM, New York, NY, USA, 183–197.

[95] Dale Skeen. 1981. Nonblocking Commit Protocols. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data (SIGMOD '81)*. ACM, New York, NY, USA, 133–142.

[96] Jeff Terrace and Michael J. Freedman. 2009. 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)*. USENIX Association, Berkeley, CA, USA, 11–11.

[97] Robbert Van Renesse, Kenneth P. Birman, Bradford B. Glade, Katie Guo, Mark Hayden, Takako Hickey, Dalia Malki, Alex Vaysburd, and Werner Vogels. 1995. *Horus: A Flexible Group Communications System*. Technical Report. Cornell University, Ithaca, NY, USA.

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

[99] Paolo Viotti and Marko Vukolić. 2016. Consistency in Non-Transactional Distributed Storage Systems. *ACM Comput. Surv.* 49, 1, Article 19 (June 2016), 34 pages.

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

[101] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. 2017. APUS: Fast and Scalable Paxos on RDMA. In *Proceedings of the Symposium on Cloud Computing (SoCC '17)*. ACM, New York, 94–107.

[102] Michael Wei, Amy Tai, Christopher J. Rossbach, Ittai Abraham, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Udi Wieder, Scott Fritchie, Steven Swanson, Michael J. Freedman, and Dahlia Malkhi. 2017. vCorfu: A Cloud-scale Object Store on a Shared Log. In *Proceedings of the 14th Conference on Networked Systems Design and Implementation (NSDI'17)*. USENIX Association, Berkeley, CA, USA, 35–49.

[103] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. 2018. Elastic Scaling of Stateful Network Functions. In *15th Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 299–312.

[104] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K. Aguilera, and Jinyang Li. 2013. Transaction Chains: Achieving Serializability with Low Latency in Geo-distributed Storage Systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 276–291.

[105] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan Ports, Ion Stoica, and Xin Jin. 2019. Harmonia: Near-Linear Scalability for Replicated Storage with In-Network Conflict Detection. (2019).