Kite: Efficient and Available Release Consistency for the Datacenter (Extended Working Version)

Vasilis Gavrielatos, Antonios Katsarakis, Vijay Nagarajan, Boris Grot, Arpit Joshi*
The University of Edinburgh, * Intel
FirstName.LastName@ed.ac.uk, * FirstName.LastName@intel.com

Abstract

Key-Value Stores (KVSs) came into prominence as highly-available, eventually consistent (EC), "NoSQL" Databases, but have quickly transformed into general-purpose, programmable storage systems. Thus, EC, while relevant, is no longer sufficient. Complying with the emerging requirements for stronger consistency, researchers have proposed KVSs with multiple consistency levels (MCL) that expose the consistency/performance trade-off to the programmer. We argue that this approach falls short in both programmability and performance. For instance, the MCL APIs proposed thus far, fail to capture the ordering relationship between strongly- and weakly-consistent accesses that naturally occur in programs.

Taking inspiration from shared memory, we advocate Release Consistency (RC) for KVSs. We argue that RC's one-sided barriers are ideal for capturing the ordering relationship between synchronization and non-synchronization accesses while enabling high-performance.

We present Kite, the first highly-available, replicated KVS that offers a linearizable variant of RC for the asynchronous setting with individual process and network failures. Kite enforces RC barriers through a novel fast/slow path mechanism that leverages the absence of failures in the typical case to maximize performance while relying on the slow path for progress. Our evaluation shows that the RDMA-enabled and heavily-multithreaded Kite achieves orders of magnitude better performance than Derecho (a state-of-the-art RDMA-enabled state machine replication system) and significantly outperforms ZAB (the protocol at the heart of Zookeeper). We demonstrate the efficacy of Kite by porting three lock-free shared memory data structures, and showing that Kite outperforms the competition.

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

Keywords Consistency, Availability, Fault tolerance, Replication, RDMA

PPoPP '20, February 22–26, 2020, San Diego, CA, USA

1 Introduction

Key-Value Stores (KVSs) came into prominence over a decade ago as highly-available, eventually consistent (EC), "NoSQL" Databases with a get/put API. Since then, they have enjoyed widespread application with use-cases that include graph applications, messaging systems, coordination services, HPC applications and deep learning [4, 13, 36, 89]. Thus, the KVS has today *transformed* into a general-purpose, programmable, distributed storage system [89], coming to resemble distributed shared memory (DSM). However, there is a key difference: unlike a traditional DSM, a KVS must also provide high availability; data must remain constantly accessible in the face of individual machine and network failures.

During the KVS' transformation, one thing became apparent: EC was no longer sufficient. Stronger consistency primitives are *essential* for achieving coordination and synchronization [8]. Indeed, in explaining why twitter's KVS was extended with strongly consistent primitives, twitter engineers concede: *"while EC systems have their place in data storage, they don't cover all of the needs of our customers"* [89].

However, strongly consistent operations invariably incur a higher performance overhead than more relaxed ones. For instance, in an asynchronous environment, implementing atomic Read-Modify-Writes (RMWs) is costlier than implementing linearizable reads/writes, which in turn is costlier than implementing weakly consistent reads/writes [7, 32, 60].

Faced with these opposing requirements, researchers have come up with a solution that now comprises the state-of-the-art: *multiple consistency level* (MCL) KVS [5, 20, 36, 54, 80, 89]. MCL KVSs enable the programmer to trade consistency for performance by requiring them to specify the consistency needs for each access. We find the MCL API unsatisfying on two grounds: programmability and performance.

- **Programmability.** The API should not ask programmers to reason about the implementation-centric consistency level for each and every access; rather it should provide them with an intuitive, programmer-centric interface.
- **Performance.** Specifying the consistency level of individual accesses fails to capture the ordering relationship between strong and weak accesses that naturally occur in programs. For example, consider the ubiquitous *producer-consumer* synchronization pattern. The producer creates an object, writing each of its 1000 fields, and then raises a flag to announce that the object is ready to be read. Meanwhile

the consumer polls on the flag; when it finally sees it raised, it proceeds to read the object. Note that the intended behavior is that when the raised flag becomes visible, the object and its 1000 fields must become visible, too. The only way to achieve this behavior in today's MCL API is to label *all* of the accesses as strong. Clearly, this is suboptimal performance-wise. An ideal API would allow for the writes to the fields to be reordered but ensure that all of these writes take effect before the write to flag.

To remedy this situation, we pose the question: Is there a consistency API that simplifies programming, while allowing for the system to extract maximum performance?

1.1 A Case for Release Consistency

To answer the question, we turn to the shared memory community which has grappled with these very questions. After a 30-year debate, the community has converged on the Data-Race-Free (DRF) programming paradigm [1] (e.g. C/C++, Java, OpenCL). DRF is a contract between the programmer and the system: if the programmer writes programs free of data races and correctly annotates synchronization operations, the system will provide strong consistency. Under the hood, the system honors the contract through a DRF-compliant memory model, typically a variant of Release Consistency (RC) [6, 27, 59, 88]. In this work, we propose the adoption of the DRF-compliant RC for distributed KVSs.

Going back to the question we posed earlier, we argue that RC ticks both boxes.

- **Programmability.** Instead of asking the programmer to reason about consistency, RC requires them to explicitly annotate synchronization operations. RC offers the typical read/write/RMW API with a twist: when writing to a synchronization variable (e.g. raising a flag or releasing a lock), that write must be marked as a *release*. When reading from a synchronization variable (e.g. testing a flag, or grabbing a lock), that read must be marked as an *acquire*.
- **Performance.** An RC enforcement mechanism can potentially leverage programmer annotations for reordering non-synchronization (*relaxed*) operations, while enforcing ordering (RC's one-sided *barrier semantics*) only when synchronization is required. However, to our knowledge the performance benefits of RC have not been explored previously in an asynchronous environment with individual machine and network failures, mainly because *there is no prior work on how to efficiently enforce RC's barrier semantics in this environment.*

1.2 Kite

In this work, we present *Kite*, the first highly-available, replicated, RDMA-enabled KVS that offers RC_{Lin}, a linearizable variant of RC (§2.3). We note that even though RC variants have been offered previously in DSM systems [18, 44, 46, 76], we are, to the best of our knowledge, the first to offer a *highly available* RC in an asynchronous setting with individual machine and network failures. In building Kite, we address three challenges:

1. Identifying protocol mappings (§3). The basic premise of RC is maximizing performance by providing strong consistency only when required. To achieve this we must identify protocols with different consistency/performance tradeoffs that map to the RC API. We identify as ideal candidates three asynchronous, fully-distributed protocols: Eventual Store (ES) [15], multi-writer-ABD [61] (an ABD [7] variant, dubbed 'ABD') and Paxos [50]. Specifically, relaxed reads and writes are mapped to ES, an efficient EC protocol that executes reads locally; releases and acquires are mapped to ABD, that offer linearizable reads and writes; and finally, RMWs are mapped to Paxos.

2. Enforcing RC barrier semantics (§4, §5). Identifying protocol mappings is not enough; the chosen protocols must be augmented to enforce RC's barrier semantics. The challenge is to do this while retaining the efficiency of ES—in particular its "local reads" property. Alas, ensuring that reads are *always* local and consistent in an asynchronous environment is challenging. Kite sidesteps this problem with a fast/slow path mechanism: the blocking *fast path* executes reads locally, albeit assuming a synchronous environment, whereas the nonblocking *slow path* can operate on an asynchronous environment, albeit sacrificing local reads. Kite alternates between the two paths. In the common case where messages are delivered on time and machines do not fail, Kite operates on the fast path. When asynchrony presents itself (e.g. through a big network delay), Kite conservatively falls back to the *slow path* temporarily, before reverting to the fast path. Thus, Kite hinges on the asynchronous slow path for progress, exploiting the synchronous fast path for performance. We describe this mechanism in Section 4 and we rigorously prove it enforces RC in Section 5.

3. Efficient system implementation (§6). We implement ES, ABD and Paxos from scratch and integrate them with Kite's slow/fast path mechanism. The salient design points of Kite are: a highly multi-threaded implementation (§6.1), the adaptation of MICA [56] (a state-of-the-art KVS) (§6.3) and the efficient use of RDMA (§6.4).

Limitations. Kite is an in-memory KVS, replicated within a datacenter for reliability, targeting the typical replication degree (ranging from 3 to 7 [36]). Therefore, Kite should not be expected to scale to hundreds of replicas. Finally, Kite does not handle replication beyond a datacenter (i.e., geo-replication) and should be combined with other systems for durability (e.g., failure of the entire datacenter).

Contributions.

- We introduce *Kite*, a replicated, RDMA-enabled KVS that offers RC_{Lin} in an asynchronous environment with network and crash-stop failures.

- Kite enforces RC's barriers efficiently via a fast-path/slow-path mechanism, that leverages the absence of failures in the common case to maximize performance, while hinging on the slow path for progress.
- Kite implements ABD, ES and Paxos and combines them with the RC barrier semantics in an RDMA-enabled, heavily multi-threaded manner.
- We rigorously prove that the fast-path/slow-mechanism of Kite enforces RC.
- Kite significantly outperforms Derecho [12] (a state-of-the-art RDMA state machine replication system) and an in-house, RDMA-enabled, multi-threaded implementation of ZAB [71] (the replication protocol at the heart of Zookeeper [36]) on a set of micro-benchmarks.
- We further demonstrate the efficacy of Kite by porting three lock-free shared-memory workloads using the Kite API, and showing that Kite outperforms the competition.

2 Preliminaries

2.1 Kite: A Replicated, Available KVS

Execution model. Kite, as is typical of replicated KVSs, is a deployment of 3-9 machines. (We use the terms machines, nodes, servers and replicas interchangeably.) Every machine holds the entire KVS *in-memory*. In each machine, a number of threads, called *workers*, execute client requests. Kite's clients use the Kite API to access its objects. A client is connected to a worker via a *session*. The order in which requests appear within a session constitutes the *session order*. Each worker is typically responsible for multiple sessions.

Failure Model. Kite assumes an asynchronous model, with network and crash-stop failures. Under this model, there is no need for synchronized clocks or bounds in message transmission delays. Individual processes (machines) might fail by crashing, but do not operate in a Byzantine manner. Network failures in either network links or messages may occur. Ensuring availability is one of the primary goals of Kite: as long as a majority of nodes (and their links) are alive, failures do not cause a disruption in Kite's operation, i.e. client requests mapped to these nodes are executed normally.

Asynchronous replication protocols. The objects of the KVS are replicated in multiple nodes to tolerate failures. An asynchronous (aka nonblocking [32]) replication protocol is then deployed to enforce consistency and fault tolerance across all replicas. A common theme across asynchronous protocols is the notion of *quorums*, which refers to a subset of the machines that hold a replica. For example a write may need to be propagated to at least a quorum of replicas before it is said to have completed. Throughout this paper, the term quorum refers to any majority of replicas.

2.2 Consistency Models

Eventual Consistency (EC). A number of different weak consistency models with various guarantees [79] are categorized as variants of EC [85], all of which mandate that replicas must converge in the absence of new updates. We identify *per-key Sequential Consistency* (per-key SC) [20, 57, 83] as an intuitive, well-defined safety variant of EC. Per-key SC mandates that: 1) all sessions agree on one single order of writes for any given key (aka write serialization) and 2) reads and writes to the same key appear to perform in session order.

Sequential Consistency (SC). SC mandates that reads and writes (across all keys) from each session appear to take effect in some total order that is consistent with session order [49]. To put it succinctly, SC enforces session ordering.

Linearizability (lin). In addition to SC's constraints, lin mandates that each request appears to take effect instantaneously at some point between its invocation and completion [33]. Thus, lin not only enforces session ordering, but also preserves real-time behavior.

2.3 Release Consistency

RC_{SC} provides a sequentially consistent variant of RC. RC_{SC} has strong enough primitives that lets one (provably) achieve well-known synchronization patterns, including wait- and obstruction-free concurrent implementations of linearizable objects, as well as mutual exclusion [8, 32].

We start the discussion with RC_{SC} and then extend it to RC_{Lin}, which is the consistency model that Kite provides. Table 1 describes the RC_{SC} API and the session orderings enforced, where p→q means that operation p appears to take effect before operation q. In Section 5, we formalize RC axiomatically.

SC semantics (release/acquire→release/acquire). RC_{SC} enforces SC among releases and acquires; i.e. releases and acquires appear to take effect in session order.

Release barrier semantics (all→release). A release acts as a one-way barrier for all prior accesses; i.e., a release takes effect only after writes and reads, before the release, take effect. Informally this means that, by the time the release write becomes visible to another session: (1) all writes that precede the release must be visible to that session and (2) all reads that precede the release must have returned.

Acquire barrier semantics (acquire→all). An acquire acts as a one-way barrier for subsequent accesses; i.e., reads and writes after the acquire, appear to take effect after the acquire takes effect. Informally, when an acquire observes the value of a release from another session: (1) a read that follows the acquire must be able to observe any write that precedes the release and (2) a write that follows the acquire must not be able to affect any read that precedes the release.

RC_{SC}/RC_{Lin} API

Command	Ordering	Kite mapping
Relaxed Read/Write	no ordering	Eventual Store [15]
Release Write	all ⇒ release release ⇒ acquire	ABD [61]
Acquire Read	acquire ⇒ all	ABD [61]
RMW	all ⇒ RMW RMW ⇒ all	Paxos [50]

Table 1. RC_{SC}/RC_{Lin} API, orderings and Kite mappings.

Barrier invariant. The two types of barriers cooperate to enforce a single invariant: *when an acquire reads from a release, the accesses that follow the acquire appear to take effect after the accesses before the release.*

Enforcing RC_{Lin}. Kite enforces a stronger variant of RC_{SC}, dubbed RC_{Lin}. RC_{Lin} shares the same API with RC_{SC} and enforces the same orderings (Table 1). The only difference is that RC_{Lin} preserves lin among releases and acquires. For example, in RC_{Lin}, if a release has completed in real-time, then any subsequent acquire in real-time (from any session) is guaranteed to observe the release's result; the same does not hold for RC_{SC}. In summary, RC_{Lin} allows Kite to offer consistency semantics that range from per-key SC to lin.

3 Setting the Stage: Kite Mappings

Kite maps three existing protocols to the RC_{Lin} API as shown in Table 1. In this section, we explain our rationale behind these choices and provide an overview of each of the three protocols. We begin with Lamport logical clocks [48], as they are a vital part of all three protocols.

3.1 Lamport Logical Clock (LLCs)

An LLC [48] is a pair $< v, m_{id} >$ of a monotonically increasing version number, v, and the id of the machine that creates the LLC, m_{id}. An LLC A is said to be bigger than LLC B, if A's version number is bigger; if their versions are equal, the machine id is used as a tie-breaker.

LLCs make it possible to generate a globally unique "time" for an event without any coordination. A machine can create a unique LLC by incrementing a local version and its own machine id. LLCs can be then leveraged to order events (e.g. serialize writes) in a distributed manner, without the need for communication or with explicit ordering points (e.g. a master node). Indeed, all three protocols employed in Kite leverage LLCs to avoid centralized points when ordering events.

3.2 Eventual Store for relaxed reads and writes

Eventual Store (ES) [15] achieves per-key SC for replicated KVSs by maintaining an LLC for every key, using which it is able to provide a unique LLC for every write, thereby serializing writes to each key.

Why ES? ES is extremely efficient, incurring no more than the absolutely necessary protocol overhead: reads execute locally and writes broadcast the new value, an action that is necessary for fault tolerance. Besides, ES is naturally asynchronous and tolerant to failures.

3.3 ABD for releases and acquires

The multi-writer-ABD algorithm [61] builds on the seminal ABD algorithm [7] to emulate linearizable reads and writes on replicated data over a message passing system, on an asynchronous environment. (For brevity, we refer to multi-writer-ABD simply as ABD.)

Why ABD? For four reasons: 1) ABD offers lin in an asynchronous environment, allowing Kite to offer lin among releases and acquires. 2) ABD is very efficient: it is fully distributed and it naturally lends itself to a multi-threaded implementation. 3) ABD is designed explicitly for full writes, which *do not require consensus*[32], thereby avoiding the implications of the consensus impossibility result [23]. 4) ABD is a natural match for ES: both protocols use broadcasts and per-key LLCs, enabling sharing of metadata and network optimizations across them. Below, we describe ABD, noting that an LLC is maintained for each key.

Write. A write request performs two broadcast rounds, gathering responses from a quorum of machines for each round. A first lightweight round that reads the per-key LLCs of remote replicas, and a second round that broadcasts the new value along with its LLC.

Read. A read request performs one broadcast round where it reads the keys and LLCs from a quorum of replicas, returning the value with the highest LLC. If the value to be returned has not been seen by a quorum of replicas, then a second broadcast is performed with that value and its LLC.

3.4 Paxos for RMWs

Paxos [50] is a state machine replication protocol that allows distributed processes to achieve consensus in an asynchronous environment amidst machine and network failures.

Why Paxos? RMWs require consensus. Out of the myriad of consensus protocols, we choose Paxos because it is a well-established protocol that allows for high-performance implementations: it can be implemented in a per-key fashion, enabling concurrency among different keys, and without leaders or centralized points that hinder availability and concurrency. Below, we first provide a brief overview of the Paxos protocol and then we describe how we incorporate Paxos in Kite to implement RMWs.

Basic Paxos operation. Paxos requires two broadcast phases: a *propose* phase and an *accept* phase. When a replica acks an accept for a Paxos command, it is said to accept the command. If a command is accepted by a quorum of replicas, then the command is said to have committed. In practice (and in

Kite), a *commit* message is also broadcast to notify the rest of the replicas. Therefore, a Paxos command in Kite typically completes within three broadcast rounds.

Per-key. Because RMWs to different keys commute, they need not be ordered [51]. This observation allows us to execute Paxos at a per-key granularity, uncovering the available request-level parallelism across RMWs to different keys and enabling a multi-threaded implementation, as threads need to synchronize only when accessing the same key.

Leaderless. Lamport proposes that when Paxos is executed repeatedly (i.e., multi-decree Paxos), it should elect a stable leader [50]. The stable leader can execute the propose phase for only the first command it commits, and avoid it for all the rest. Nonetheless, Kite implements leaderless Basic-Paxos [50], similarly to [72]. In doing so, we concede the extra round-trip per RMW, but we maintain the properties that made us choose Paxos in the first place: the constant availability, and the concurrent/decentralized nature of the protocol.

3.5 Carstamps

ABD and ES writes naturally serialize through the use of LLCs. In order to serialize Paxos RMWs with ABD/ES writes we use *carstamps* [16]. A carstamp is a pair $< LLC, paxos-no >$, of an LLC and a monotonically increasing number that denotes how many times we have executed Paxos on a key. A carstamp C1 is bigger than C2, if C1's LLC is greater, or if their LLCs are equal and C1's paxos-no is greater. An RMW will select an LLC as its *base*. That is, it will select a unique ABD/ES write to serialize after. That allows all machines to agree on a single order between ABD/ES writes and Paxos RMWs.

4 Enforcing RC Barrier Semantics

In the previous section, we described how Kite maps the RC API to existing protocols. This is not sufficient to enforce RC barrier semantics, however. Kite enforces the barrier semantics through its fast/slow path mechanism, relying on a nonblocking slow path for progress, while leveraging a blocking fast path for performance. We first provide the big picture, explaining the problem that the mechanism addresses and its solution (§4.1). We then provide an in-depth description of the mechanism (§4.2) and discuss its optimizations (§4.3).

4.1 Big picture

Consider the example shown in Figure 1, assuming that sessions, S1 and S2, are mapped to different machines. (For brevity, we refer to the machines using the session names.) RC mandates that if S2's read of $flag$ (acquire) returns 1, then its read of X must also return 1. Since relaxed reads in Kite are mapped to ES, they are performed locally. Therefore, to enforce RC, Kite must ensure that S1's write to X reaches S2 before the write (release) to $flag$.

Fast path: RC & ES without asynchrony. In the common case where machines operate without big delays, the condition is met in Kite through the *fast path* which enforces one simple rule: before the release begins its execution, Kite ensures that each write prior to the release is acked by *all* replicas. This rule enables a relaxed read to execute locally without violating RC. by the time the acquire from S2 returns $flag = 1$, S2 must have already acked the write to X, and thus can execute its read to X locally via ES.

The problems caused by asynchrony. Alas the fast path rule that requires each write before a release to be acknowledged by *all* replicas cannot be enforced in an asynchronous environment. For instance, assume that S1 does not receive an ack from S2 for the write to X. The ack may have not arrived because S2 has failed or because S2 is slow. That presents S1 with a dilemma: on the one hand, if S2 has failed, S1 should not block indefinitely waiting for an ack; on the other hand, if S2 is alive, S1 should wait for its ack or risk S2 reading $X = 0$. Even worse, if S2 is alive but has simply missed the write from S1, S1 can neither wait, as it will block indefinitely, nor move on, as it will violate RC.

Kite's solution: The fast/slow path. Kite solves this problem through its fast/slow path mechanism: on an acquire, S2 discovers whether it has lost a write message. If so, S2 deems its entire local storage to be stale (*out-of-epoch*), transitioning itself to the *slow path*, where it must refresh each of the keys before accessing them again locally (i.e. with ES). Note that, unless S2 performs another acquire, it only needs to refresh each key once, because in RC, the relaxed accesses need only be as fresh as the latest acquire.

While rendering the entire local storage stale may appear as an extreme measure, we note that this overhead is rarely incurred, because in a controlled, datacenter environment, asynchrony is relatively rare [11, 45]. More importantly, shifting all the overhead to the misbehaving machine allows for a very efficient fast path, as it ensures that *asynchrony-related overheads are incurred only when asynchrony manifests.*

Below we sketch how the fast/slow path mechanism will work for the example in Figure 1.

$\succ$ **On a release.** Before writing to $flag$ (release), S1 attempts to gather acks from all machines for its write to X within a timeout. If the timeout expires and S1 has not received an ack from S2, then S1 first broadcasts that S2 is *delinquent* (i.e., is suspected to have missed one or more writes), ensures that a quorum of machines have been informed of S2's delinquent status, and then finally, proceeds with its release.

$\succ$ **On an acquire.** Because acquires are implemented with ABD, when S2 acquires $flag = 1$ at a later time, it must reach a quorum of machines and thus will intersect with the quorum that knows of S2's delinquent status. Then, and before completing the acquire, S2 renders its entire local store stale

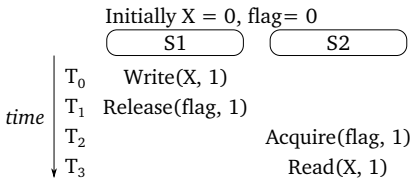


Figure 1. Producer-consumer pattern between S1 and S2.

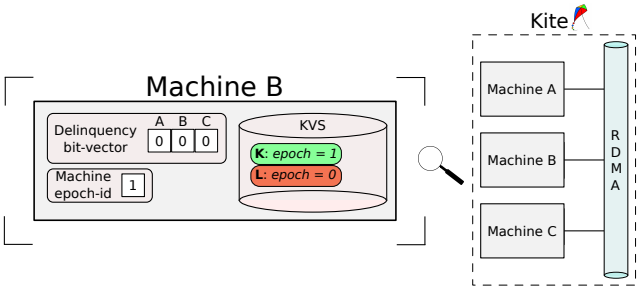


Figure 2. Zooming inside Machine B. Key L is out-of-epoch (slow-path); key K is in-epoch (fast-path).

(out-of-epoch), by simply incrementing its *machine epoch-id*. (The epoch semantics is described in the next section.)

➢ **On a relaxed access.** A relaxed access to an out-of-epoch key cannot be performed with ES (i.e. in the fast path). Instead, the key is restored *in-epoch* in the slow path, through an ABD access (i.e. a stripped-down ABD as explained in §4.3). A key is restored by simply advancing its own key's *epoch-id* to match the machine's epoch-id.

One final problem. After S2 transitions to the slow path, it must notify the remote machines that it has been made aware of its delinquent status and has transitioned to the slow path. This is necessary to prevent the pathological case where subsequent acquires from S2 keep discovering that S2 is delinquent, needlessly bringing it back to the slow path. However, restoring its status as non-delinquent in remote machines is not a trivial action, as S2 must ensure that the status is restored atomically and after it has transitioned to the slow path. We defer the discussion of how Kite achieves the task for Section 4.2.1.

4.2 Kite's fast/slow path mechanism

This section provides an in-depth description of the fast/slow path mechanism.

Release. Before a release can execute, it attempts to gather acks (from all machines) for each prior write in session order.

➢ **Fast-path release.** If all prior writes have been acked by all machines, the release simply executes.

➢ **Slow-path release.** If any preceding write has not been acked by all machines within a time-out, then each machine that has not acked one or more of the writes is deemed *delinquent*; we refer to the set of delinquent machines detected upon a release as *DM-set*. Before the release begins executing it enforces two invariants: (1) all previous writes have been acked by at least a quorum of machines and (2) the DM-set is known to at least a quorum of machines. To satisfy (2), a *slow-release* message is broadcast, containing the DM-set. The release begins executing only after a quorum of machines have acked the *slow-release* message.

Acquire. On an acquire, a machine learns whether it has been deemed delinquent by querying a quorum of machines (piggybacking on top of ABD read protocol actions).

➢ **Fast-path acquire.** If no remote machine deems the acquirer delinquent, the acquire barrier is enforced by simply blocking the session until the acquire has completed.

➢ **Slow-path acquire.** If the machine discovers it has been deemed delinquent, it performs the following actions: (1) blocks the session until the acquire completes and (2) transitions to the slow path by incrementing its machine *epoch-id*, rendering all locally stored keys *out-of-epoch*.

Epochs. As shown in Figure 2, each machine holds one epoch-id. (Epoch-ids of different machines are not interrelated.) Additionally, each key stores a *per-key epoch-id* as part of its metadata. Both per-key and machine epoch-ids are initially set to 0 and are monotonically increasing. On each relaxed access, the per-key epoch-id is compared against the machine epoch-id. If the key's epoch-id matches the machine's epoch-id, the key is *in-epoch* and can be accessed in the fast-path (i.e. with ES). Otherwise, if the machine epoch-id is greater, the key is said to be *out-of-epoch*, where it can only be accessed in the slow path (i.e. with ABD).

Returning to fast path. The transition to the fast path happens at a per-key granularity. Upon accessing an out-of-epoch key (in the slow path), the key's epoch-id is advanced to the machine's epoch-id, bringing the key back in-epoch. As an example, Figure 2 depicts the state of Kite machine *B*. *B*'s machine epoch-id is 1, which means it has been delinquent in the past. *B* has two locally stored keys: *L*, which is out-of-epoch and thus accessible only in the slow-path, and *K*, which is in-epoch and thus has been accessed in the slow path once, after *B* transitioned to the slow path. Note that if the machine epoch-id is incremented while a slow-path access is executing, then, when the slow-path access completes, the key must not be restored back in-epoch. For this reason, the key's epoch-id is advanced to what the machine epoch-id was when the access *started*, rather than to the value of the machine epoch-id when the access completes.

Enforcing the Barrier Invariant. Before executing a release one of the following must have happened: 1) all previous writes have been acked by all; or 2) all previous writes have been acked by a quorum, and a quorum of machines have seen

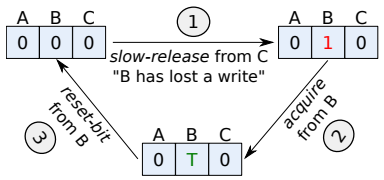


Figure 3. The transitions of the delinquency bit-vector of machine A, in a configuration with 3 machines: A, B, and C.

the DM-set. Therefore, an acquire that reads from a release, either is guaranteed to have seen all preceding writes or is guaranteed to find out about being delinquent and perform subsequent relaxed accesses in the slow path. We prove this rigorously in the Section 5.

RMWs. The discussion naturally extends to RMWs: release barrier semantics are implemented identically to regular releases and acquire barrier semantics are implemented identically to acquires.

Time-out and Availability. Recall that before a release executes, it attempts to gather all acks for prior writes within a time-out; if unsuccessful, it executes the slow path barrier. We note that increasing the length of the time-out can affect availability, but decreasing the time-out can only affect performance, as it will only mean machines go to the slow path more often. Therefore the time-out length offers a trade-off between availability and performance, and should be tuned with respect to the system requirements and the system environment. We revisit the time-out's effect in Section 8.4.

4.2.1 Setting and resetting delinquency

In a Kite deployment, each machine maintains a *delinquency bit-vector* with a *delinquency bit* for each remote machine. The delinquency bit denotes whether a given remote machine has been deemed delinquent and is used to notify that machine when it performs an acquire.

Setting a bit. Delinquency bits get set upon receiving a slow-release message. Figure 3 illustrates the transitions of A's bit-vector in a deployment with machines A, B and C. Firstly, A sets the bit for B in its bit-vector, when it receives a *slow-release* message from C, denoting that B is delinquent.

Resetting a bit. Eventually, when B executes an acquire, it reaches A, finding out that it must transition to the slow-path. At this point, A must reset its bit for B, so that subsequent acquires from B will not revert B to the slow path again. However, receiving an acquire from B is not enough for A to reset the bit; rather, A must know that B has transitioned to the slow path. To resolve this issue, when an acquirer discovers its delinquency, it broadcasts a *reset-bit* message only after it has transitioned to the slow path.

Atomic reset. Given that resetting a delinquency bit is a two-step process (acquire and reset-bit), we must ensure the bit is atomically read and reset, without any intervening slow-release messages. We ensure atomicity as follows. Each acquire is tagged with a unique id, which is included in the generated *reset-bit* message. Upon receiving the acquire from B, A transitions its bit to a transient state T and notes the unique id of the acquire. Upon receiving a *slow-release* message that marks B as delinquent, A unconditionally sets B's bit to 1. Upon receiving a reset-bit message, A transitions the bit back to 0, iff the bit is still in T state and the reset-bit originates from the acquire that transitioned the bit to T.

4.3 Optimizations

Having established how Kite enforces RC, we now describe two non-intrusive, protocol-level optimizations.

Overlapping a release with waiting. The first broadcast round of a release (i.e., ABD write) reads the LLCs from a quorum of machines for the key to be written, to ensure that the releaser uses a sufficiently big LLC. Because reading remote LLCs is a benign action that does not notify remote machines of the ensuing release, we perform it early, overlapping its latency with waiting for acks of prior writes. Extending to the RMWs, we overlap waiting for acks with the Paxos first phase (i.e., proposing), which, similarly to the first round an ABD write, does not contain the new value to be written.

Slow-path optimization. We earlier specified that the slow path of relaxed reads and writes is implemented with ABD. However, ABD provides more guarantees than required in this instance, as it is fully linearizable, whereas we only seek to enforce RC. Specifically, the slow-path must ensure that a relaxed read observes any completed relaxed write that may have been missed, and as such, it is sufficient to read from a quorum of machines, guaranteeing an intersection with writes. Therefore, the optional second round broadcast of ABD reads is not required in this instance, as relaxed reads need not make sure that the read value has been seen by a quorum. In the same spirit, we complete writes without waiting for acks, as relaxed writes need not ensure that the write has been seen by a quorum; rather the subsequent release in session order is responsible for that.

5 Proof: Kite's fast/slow path enforces RC

In this section, we prove informally that Kite's fast/slow path mechanism enforces RC. We first specify RC (§5.1) and provide a high-level sketch of the proof (§5.2). Then, we identify the different cases of Kite's operation, proving correctness on a case-by-case basis. Specifically, we focus on three cases: Kite's fast-path (§5.3), the transition from fast-path to slow-path (§5.4) and finally the transition from slow-path to fast-path (§5.5).

5.1 Release Consistency Semantics

We use the following notation for memory events:

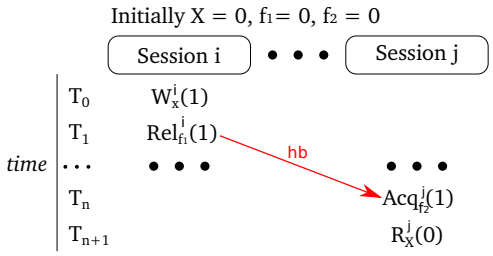


Figure 4. Proof sketch assumed violation. The RC violation is that Session A reads $X = init$, instead of $X = 1$.

- $\mathbf{M^i_x}$: memory operation (any type) to key x from session i. The operation can be further specified as a read: R^i_x, a write W^i_x or with an identifier (e.g. $M1^i_x$)
- $\mathbf{Rel^i_x}$: a release (release write or release-RMW) to key x from session i.
- $\mathbf{Acq^i_x}$: an acquire (acquire read or acquire-RMW) to key x from session i.

Note that our RMW reads have acquire semantics and RMW writes have release semantics automatically. We use the following notation for ordering memory events:

- $\mathbf{M^i_x} \xrightarrow{so} \mathbf{M^i_y}$: $\mathbf{M^i_x}$ **precedes** $\mathbf{M^i_y}$ in session order.
- $\mathbf{M^i_x} \xrightarrow{hb} \mathbf{M^j_y}$: $\mathbf{M^i_x}$ **precedes** $\mathbf{M^j_y}$ in the global history of memory events, which we refer to as happens-before order ($\xrightarrow{hb}$).

We formalize Release Consistency using the following rules:

i) A memory access that precedes a release in session order appears before the release in happens-before: $M^i_x \xrightarrow{so} Rel^i_y \Rightarrow M^i_x \xrightarrow{hb} Rel^i_y$.
ii) A memory access that follows an acquire in session order appears after the acquire in happens-before: $Acq^i_y \xrightarrow{so} M^i_x \Rightarrow Acq^i_y \xrightarrow{hb} M^i_x$.
iii) An acquire that follows a release in session order appears after the release in happens-before: $Rel^i_y \xrightarrow{so} Acq^i_x \Rightarrow Rel^i_y \xrightarrow{hb} Acq^i_x$.
iv) Two memory accesses to the same key ordered in session order preserve their ordering in happens-before: $M1^i_x \xrightarrow{so} M2^i_x \Rightarrow M1^i_x \xrightarrow{hb} M2^i_x$.
v) RMW-atomicity axiom: an RMW appears to executes atomically, i.e., for an RMW that is composed of a read R^i_x and a write W^i_x, there can be no write W^j_x such that $R^i_x \xrightarrow{hb} W^j_x \xrightarrow{hb} W^i_x$.
vi) Load value axiom: A read to a key always reads the latest write to that key before the read in happens-before: if $W^j_x \xrightarrow{hb} R^i_x$ (and there is no other intervening write W^k_x such that $W^j_x \xrightarrow{hb} W^k_x \xrightarrow{hb} R^i_x$), the read R^i_x reads the value written by the write W^j_x.

5.2 Proof Sketch

The key result that needs to be proved is that Kite enforces the load value axiom: a read must return the value written by the most recent write before it in happens-before. Below, we provide a sketch of a proof, identifying the non-trivial cases that need to be proved more rigorously, along the way.

One degenerate case is when both the write and the read are from the same session. In this case, the load value axiom is enforced since Kite honors dependencies within each session. More specifically, in the fast path, the write would have been applied to the KVS before the read performs. In the slow path, every read explicitly checks for dependencies with previous writes in progress.

Therefore, the interesting case is when the write and read are from two different sessions: specifically W^i_x (from session-i) and R^j_x (from session-j). Without loss of generality we assume that W^i_x and R^j_x are relaxed operations. The fact that the write appears before the read in happens-before implies that there must be a release after the write in session-i and an acquire before the read in session-j, such that the release is ordered before the acquire in happens-before. As shown in Figure 4, given that $W^i_x \xrightarrow{so} Rel^i_{f_1} \xrightarrow{hb} Acq^j_{f_2} \xrightarrow{so} R^j_x$, we need to prove that R^j_x returns the value written by W^i_x (i.e. $X = 1$), and not the previous value of X (i.e. $X = 0$).

We first prove the following lemma and then proceed to our proof by examining the different cases. For simplicity, for the rest of the section, we omit the thread identifiers from the memory operations of Figure 4, referring to them as W_x, Rel_{f_1}, Acq_{f_2} and R_x.

Lemma 5.1. *Acq_{f_2} cannot complete its execution (in real time) before Rel_{f_1} begins execution.*

Proof. $Rel_{f_1} \xrightarrow{hb} Acq_{f_2}$ implies that Acq_{f_2} (in the general case) is at the end of a happens-before chain of releases and acquires and Rel_{f_1} is at the top of this chain. Because releases and acquires are linearizable in Kite (owing to ABD), Acq_{f_2} cannot complete its execution before Rel_{f_1} begins execution. □

5.3 Case 1: Fast-path (no failures or delay)

Let us assume that both session-i and session-j are operating in fast-path. (I.e., the machines in which the sessions are mapped are operating in the fast-path.) Kite ensures the load value axiom via the following real-time orderings:

- Before executing a release, Kite waits for all prior writes to be acked by all. This means that W_x is acked by session-j before Rel_{f_1} begins.
- Acq_{f_2} cannot complete its execution (in real time) before Rel_{f_1} begins execution (from Lemma 5.1).
- Kite blocks the acquiring session until the acquire completes. This means that R_x begins execution only after the acquire Acq_{f_2} completes.

The above real-time orderings imply that R_x begins execution only after W_x has been acked by session-j, and hence will read the correct value.

5.4 Case 2: Fast-path/Slow-path transition (failure or delay)

Both sessions are initially operating in the fast-path, but session-j fails to receive the write, W_x, owing to a failure (e.g., a message delay). In this case, the read R_x must still return the value written by the write, and thus cannot execute locally in the fast-path.

To this end, we must ensure the following. First, session-i should detect that session-j is delinquent (i.e., suspected to have missed a write) and must broadcast this information. Second, when session-j performs its acquire, it must discover it has been deemed delinquent and must transition into the slow path. Finally, when session-j transitions to the slow-path, its read to X must read session-i's write to X. From the above we can infer the following three lemmas that must be enforced in Kite for the load value axiom to hold.

Lemma 5.2. *Before executing a release, the set of delinquent machines (DM-set) must be identified and, if not empty, broadcast to a quorum.*

Proof. This is enforced by Kite's actions for a release. Kite attempts to wait for all writes that precede a release to gather acks from all replicas before executing a release. If not all acks can be gathered, the DM-set will be broadcast and the release will not begin executing until the DM-set broadcast is acked by a quorum of machines. □

Lemma 5.3. *For a release Rel_{f_1} and an acquire Acq_{f_2}, with $i \neq j$, and $Rel_{f_1} \xrightarrow{hb} Acq_{f_2}$, and if Rel_{f_1} happens to publish delinquent machines before its execution, then Acq_{f_2} should be able to read the set of delinquent machines published.*

Proof. A release writes a new value to a quorum of replicas. Before any replica is updated with the released value, the DM-set would have already reached a quorum of replicas. It follows that *if the released value can be seen, the DM-set has reached a quorum of replicas*. This is the *release invariant*.

Case a: the release synchronizes with the acquire. I.e., the acquire Acq_{f_2} reads the value of release Rel_{f_1}. (This is only possible if $f_1 = f_2 = f$). Following ABD, an acquire gathers responses from a quorum of replicas, and reads the the value with the highest LLC. If it cannot ensure that the read value has been seen by a quorum, it broadcasts a write with the value. There are two cases: 1) if Acq_f reads the value of Rel_f from a quorum of replicas, the quorum of replicas that replied with the new value must intersect with the quorum that has seen the DM-set (because of the *release invariant*), and therefore Acq_f is guaranteed to see the DM-set in the intersection replica. 2) if Acq_f reads the value of Rel_f from fewer than a quorum of machines, then Acq_f will include a second broadcast round to write the value. In that case, it is guaranteed that the second broadcast round of Acq_f will begin only after the value of Rel_f has been written to at least one replica (which can only happen after the DM-set has reached a quorum, i.e. *release invariant*), and thus the quorum of replicas reached by the second round of Acq_f must intersect with the quorum of machines that have seen the DM-set.

Case b: the release does not synchronize with the acquire. I.e., Acq_{f_2} does not read from Rel_{f_1}. However, $Rel_{f_1} \xrightarrow{\text{hb}} Acq_{f_2}$ implies that Acq_{f_2} is at the end of a synchronization chain of releases and acquires and Rel_{f_1} is at the top of that chain; that chain must include a release/acquire that saw the value written by Rel_{f_1}, and only after it had seen that value (and thus after the DM-set has reached a quorum of replicas), it created a new value f_2 that was read by Acq_{f_2}. Therefore, it follows that by the time the value f_2 can be read, the DM-set has already reached a quorum of replicas. The rest of the proof then follows the same structure as when the acquire reads from the release (i.e., case a). □

Lemma 5.4. *If an Acq_{f_2} of session-j discovers itself to be delinquent, then the next relaxed access to key X will happen in the slow path.*

Proof. A key X is accessed in the fast-path, iff the epoch-id of key X is equal to the machine's epoch-id. If X's epoch-id is smaller than the machine's epoch-id then X can only be accessed in the slow-path. Accessing X in the slow-path will advance X's epoch to what the machine's epoch-id was, when the slow-path access to X was initiated. Therefore, X's epoch-id can never be bigger than the machine's epoch-id, as the machine's epoch-id is monotonically incremented, and X's epoch-id only gets modified to match a snapshot of the machine's epoch-id.

Now assume that an acquire Acq_{f_2} discovers it has been deemed delinquent and thus it increments the machine's epoch-id (transitioning to the slow path) before completing the acquire at time T_1. It follows that at time T_1, the machine's epoch-id is bigger than X's epoch-id, because X's epoch-id can only be advanced to the newly incremented epoch-id, if it is accessed in the slow-path after time T_1. Therefore, if session-j issues a relaxed access to X after Acq_{f_2}, then it must be that X's epoch-id is smaller than the machine's epoch-id, and thus X will be accessed in the slow path. □

Having proved the lemmas above, we are now in a position to prove the load-value axiom.

Lemma 5.5. *For a write W_x, release Rel_{f_1}, acquire Acq_{f_2} and a read R_x such that: $W_x \xrightarrow{so} Rel_{f_1} \xrightarrow{hb} Acq_{f_2} \xrightarrow{so} R_x$, and if there is no intervening write to X between W_x and R_x, R_x will read the value written by W_x.*

Proof. First, we observe that Acq_{f_2} cannot complete execution before Rel_{f_1} begins execution. (from Lemma 5.1). Then, we observe that since $W_x \xrightarrow{\text{so}} Rel_{f_1}$, it implies that at least a

quorum of acks for W_x must have been gathered before Rel_{f_1} begins execution. In a similar vein, since $Acq_{f_2} \xrightarrow{so} R_x$, Kite ensures that R_x does not begin execution until after Acq_{f_2} has completed. Therefore, Kite must have gathered at least a quorum of acks for W_x, before R_x begins execution. Therefore, this means that: if R_x executes in the slow path it is guaranteed to read the value of W_x.

If R_x executes in the fast path, then it must be that W_x gathered an ack from the machine that R_x executes from. On the other hand, if W_x could not gather an ack from the machine that R_x executes from, then from Lemmas 5.2, 5.3, 5.4, it follows that Rel_{f_1} will have detected the DM-set and Acq_{f_2} will have discovered its delinquency transitioning into the slow-path and thus the R_x would happen in the slow path and would be hence guaranteed to read the value of W_x. □

5.5 Case 3: Slow-path/Fast-path transition

Once a session goes into the slow-path and reads a key using ABD, Kite allows subsequent relaxed accesses to that key to execute in the fast-path. This is safe since RC requires only that new values must be seen upon an acquire. As we already saw in case 2, upon encountering an acquire, the acquiring session is guaranteed to learn about its delinquency and increment its machine epoch-id, rendering all locally stored keys out-of-epoch and thus guaranteeing that the next access to every key will happen in the slow path.

When an acquire discovers its delinquency, it attempts to reset the delinquency bits in remote machines, so that subsequent acquires need not be notified again for the same missed messages. Thus, resetting delinquency bits is a best-effort approach to prevent repeated redundant transitions to the slow path. To ensure correctness, we must guarantee that the acquirer never resets a bit in a manner that can cause a consistency violation. We identify two invariants necessary for safety and prove that they are enforced.

First, a delinquency bit for a machine can be reset only after the machine has transitioned into the slow path, i.e., only after its epoch-id has been incremented. Otherwise, another racing acquire from the same machine (but different session) could find the bit reset and go on to erroneously access a local key in the fast-path. Second, a delinquency bit must be reset atomically by the acquire, i.e., between the time when the session performs the acquire and resets the bit, the machine must not have lost a new message. From the above, we infer the following two lemmas that must be enforced by Kite.

Lemma 5.6. *A delinquency bit for a machine is reset only after the epoch-id of the machine has been incremented.*

Proof. This is enforced by Kite's actions. When an acquire discovers that the machine is delinquent, it broadcasts a *reset-bit* message only after incrementing its machine epoch-id. □

Lemma 5.7. *A delinquency bit that was observed by acquire Acq_x will be reset iff there has been no attempt to set the bit (by a racing slow-release) in between receiving Acq_x and its spawned reset-bit message.*

Proof. Recall from § 4.2.1, that an acquire, upon detecting a set delinquency bit, it transitions it to state T and tags it with its unique-id. Additionally, reset-bit messages *carry* the unique ids of their parents. When a reset-bit message is received, it resets the delinquency bit iff the bit is in state T and the carried unique-id matches that of the bit. On resetting a bit, all written unique ids are cleared. Finally, when receiving a slow-release message, the relevant delinquency bits are unconditionally set to 1. Therefore, any subsequent reset-bit message will be disregarded. □

Remark. *A delinquency bit can be detected by multiple acquires as each machine can run many concurrent sessions, but each session can only have one outstanding acquire at any given moment, as acquires block the session. Therefore, the number of unique-ids that may need to be stored with each delinquency bit is bounded by the number of sessions that can run on a Kite machine.*

Remark. *The transient state T is not essential, as the clearing of all unique ids of a bit on receiving a slow-release would have the same effect. Rather, state T is used for convenience, as it simplifies the actions of resetting and setting a delinquency bit.*

6 System Design

In this section, we provide a brief overview of Kite's implementation. Firstly, we provide a functional overview of Kite along with its API (§6.1) and then we provide details on Kite's KVS (§6.3) and its networking (§6.4). The source code of Kite is available online [25].

6.1 Functional Overview and API

A Kite node is composed of client and worker threads. Client threads use the Kite API to issue requests to worker threads, which execute all of the Kite actions to complete the requests.

Client Threads. The client threads can be used in two ways: 1) clients of Kite can be collocated with Kite, in which case the client threads implement the client logic and 2) clients can issue requests remotely, in which case client threads act as a mediator, propagating the requests to the worker threads. For the rest of the paper we assume that clients are collocated with Kite and we simply refer to the client threads as clients.

Worker Threads. Worker threads (or simply *workers*), are the backbone of Kite, as they execute the client requests by running the three protocols, honoring the RC semantics and maintaining the KVS. Each worker is allocated a number of client sessions, executing only their requests. To avoid unnecessary synchronization among workers, each session is allocated to exactly one worker. Finally, a worker is connected with exactly one worker in each remote machine, exchanging the necessary protocol-level messages to execute requests.

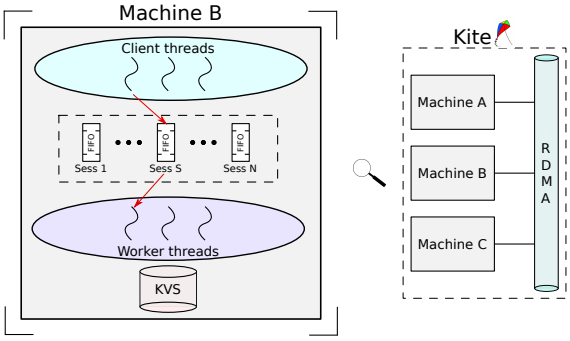


Figure 5. A Kite machine is composed of worker and client threads, that interface through the session FIFOs.

Designed for parallelism. In designing Kite we focus on achieving very high throughput by uncovering all available parallelism across unrelated requests, i.e., *request-level parallelism*. Specifically, Kite uncovers 1) the thread-level parallelism across workers threads (which only need to synchronize when accessing the same key), 2) session-level parallelism within a worker thread (as every worker is typically responsible for multiple sessions) and finally, 3) the parallelism allowed by RC within one session, overlapping the execution of same-session relaxed accesses.

6.2 Kite API.

Clients interface with workers through sessions. The client is assigned a session, which it uses on every call to the Kite API. Under the hood, Kite statically maps sessions to workers, and maintains one FIFO queue per session (called *session FIFO*). Upon issuing a request, the client provides its assigned session and the Kite runtime inserts the request to the corresponding session FIFO. The worker that is responsible for that session picks up the request and completes it. Note that the FIFO nature of the queues implements the session order: the order in which a client issues requests for a session constitutes the session order. Finally, the client is notified of the request's completion either by blocking, when using the *sync API* or by polling at a later time, when using the *async API*.

The Kite API offers relaxed reads/writes, release-writes, acquire-reads, a Fetch-&-Add (FAA), and two variants of Compare-&-Swap (CAS): a weak variant that can complete locally if the comparison fails locally, and a strong variant that always checks remote replicas. The Kite API includes an asynchronous (*async*) and a synchronous (*sync*) function call for every request (similarly to Zookeeper [36]).

Synchronous API . A sync call issues the request and then blocks polling for the request's completion. We provide here the function call that issues a sync relaxed read:

```
1 sync_read(key_id, val_len, *read_value_ptr,
      ↪session_id)
```

The programmer provides the key to be read (*key_id*), the size of the value in bytes (*val_len*), a pointer where the value should be copied (*∗read_value_ptr*) and the session id (*session_id*). The call returns an integer, which, ifnegative, maps to an error code. Sync calls simplify programming, but are not very efficient, as the client may need to block for several microseconds waiting for a request to complete.

Async API . An async call returns immediately before the request has completed. The client can call a polling function to find out if the request has been completed. As an example, we provide here the async relaxed read call:

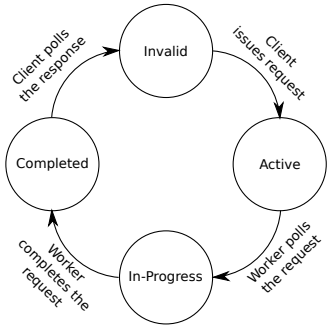
```
1 async_read(key_id, val_len, *read_value_ptr,
      ↪session_id)
```

The call returns an integer, which, if negative, maps to an error code; otherwise, the returned integer denotes the *request id* that can be used by the client to poll for the request's completion. Kite provides a range of polling functions, that typically require a session id and a request id as arguments.

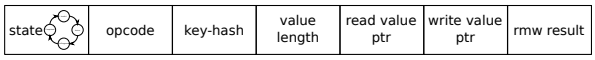
Batched Asynchronous Programming . Despite its performance benefits, an asynchronous API is admittedly quite cumbersome to program with. For that reason, we make the following simplification: completed requests can only be polled in session order, irrespective of the order in which the worker completes them. This enables the client thread to issue a batch of requests and then at a later time, poll only for the last request issued. If the last request is successfully polled, it guarantees that all preceding requests have been completed. We found this pattern very natural in porting code to Kite

Multiple sessions per client thread. A client thread can use multiple sessions to improve performance: enabling thread-level parallelism across the workers, and session-level parallelism within one worker thread. Programmers can leverage this feature to parallelize their applications, by allocating parallelizable tasks to different sessions. We leverage this capability when porting lock-free data structures to Kite, in order to allow clients threads to work on multiple distinct operations concurrently, through different sessions.

Session FIFO. Session FIFOs constitute the communication medium between client and worker threads. There can be thousands of sessions FIFOs (one per session), where each session maps to exactly one client and one worker thread. Therefore, any given session FIFO can only be accessed by one worker and one client. We focus on one slot of a single session FIFO. The slot's fields are illustrated in Figure 6a. The client fills the fields of the slot to issue a request, and the worker uses the fields to complete the request. For instance, on a CAS request the worker writes the result in the *rmw result* field. If the CAS is unsuccessful, the worker also writes the read value in the address pointed to by the *read value ptr* field.



b) The FSM of the *state* field



a) A single slot of a session

Figure 6. The fields of one slot of one session FIFO, and the FSM of the state field.

Request FSM. A FIFO slot contains a *state* variable, which is used to facilitate the synchronization between worker and client. The state variable works as an Finite State Machine (FSM) (Figure 6), transitioning between four possible states, denoting who can access the slot. A client issues a request to the slot only if the state is *Invalid*; issuing the request transitions the state to *Active*, which implicitly passes the ownership of the slot to the worker thread. The worker will transition the slot to *In-progress* when it polls it and later to *Completed* when it completes it.

6.3 Key-Value Store implementation

Every node in Kite maintains a local KVS. The implementation of the KVS is largely based on MICA [56] as found in [42], with the addition of sequence locks (seqlocks) [47], from [26], to enable multi-threading.

Adapting MICA. The MICA API cannot capture the full depth of all ES, ABD, Paxos actions (e.g., . proposes/accepts). We rectify this, by first creating a new structure for the MICA key-value pair that also contains the seqlock, the state of the key (i.e., proposed, accepted or committed) the carstamp, the last accepted value, the highest proposed LLC, highest accepted LLC, a unique id for the last committed RMW and the unique id of the RMW that is currently working on the key, if the key is in proposed/accepted state.

Finally, we have modified MICA, such that it reverts to our handlers for all the actions of the three protocols.

6.4 Network Communication

Kite adopts the RDMA paradigm of Remote Procedure Calls (RPCs) over UD Sends, that has been shown to be a practical, high-performance design [26, 41–43].

RDMA Optimizations. We carefully implement low-level, well-established RDMA practices, such as doorbell batching and inlining, (reader is referred to [10, 26, 42, 43] for a detailed explanation). Additionally, we minimize the number of network connections to alleviate network metadata pressure from CPU and NIC caches and TLBs, by connecting each worker to exactly one worker of each remote Kite machine.

Network Batching. The RPC paradigm enables batching multiple messages in the same network packet. Kite worker threads leverage this capability, batching messages in the same packet opportunistically: workers never wait to fill a quota, rather they form a packet from available messages. Opportunistic batching has a significant impact in performance, as the overhead of both network and DMA transactions is amortized (i.e., network headers, PCIe headers etc.). Additionally, batching across all protocols facilitates combining the implementation of common functionality.

Broadcasts. Finally we note that all three protocols contain broadcast primitives. We implement broadcasts through unicasts in the same manner as [26].

7 Methodology

A baseline system for Kite should be an RDMA-enabled, replicated KVS that operates in an asynchronous environment amidst crash-stop and network failures. In an effort to identify existing systems that fulfill these requirements, we compare against two systems:

1. Derecho (open-source). We identify Derecho [38] as the most efficient amongst a series of RDMA State Machine Replication implementations [38, 70, 87]. We use the open-source implementation for our evaluation.

2. ZAB (in-house). Zookeeper Atomic Broadcast (ZAB) [71] is the replication protocol at the heart of Zookeeper [36]. We implement ZAB over a replicated KVS (same as Kite), RDMA-enabled and multi-threaded, applying all Kite optimizations. Our ZAB outperforms the open-source implementation of Zookeeper (evaluated in [39]) by three orders of magnitude. ZAB enforces orderings by specifying a total order across all writes; all nodes apply the writes in that order. This approach allows ZAB to perform SC reads locally.

Infrastructure. We conduct our experiments on a cluster of 5 servers interconnected via a 12-port Infiniband switch (Mellanox MSX6012F-BS). 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 PCIe-gen3 x16) connected on socket 0. 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).

Workloads. Similarly to prior work [36], we use KVS workloads with reads and writes, including releases, acquires and

RMWs, for Kite. The KVS consists of one million key-value pairs, which are replicated in all nodes. We use keys and values of 8 and 32 bytes, respectively which are accessed uniformly. For Kite, requests are issued from its client threads over the async API. As application examples, we implement and evaluate three lock-free data structures over Kite API.

8 Evaluation

8.1 Throughput overview of the protocols

Figure 7 shows the performance of Kite and ZAB, while varying the write ratio from 1% through 100%. Because Kite is composed of three different protocols—ES, ABD, Paxos—Kite's performance is bounded by those. To better understand where Kite falls within the boundary, we also compare against each of these constituent protocols. (Derecho is omitted from this experiment as we were unable to vary its write ratio.) Below, we discuss each of the protocols, highlighting the performance, in million requests per second (mreqs), at 1% and 100% write ratios.

ES: 765 **to** 96 **mreqs .** ES provides per-key SC. Because reads are always local in ES, ES serves as an upper bound for Kite. Because writes in ES requires a broadcast, its throughput drops with increasing write ratios.

ABD: 130 **to** 62 **mreqs.** ABD offers linearizable reads and writes, but not consensus (i.e., it does not support RMWs). ABD serves as the lower bound of Kite when all accesses are marked as synchronizing, but none of them are RMWs.

ZAB: 172 **to** 16 **mreqs.** By totally ordering writes, ZAB provides RMW semantics for its writes, but relaxes the consistency of reads to allow for local reads. We observe that ZAB outperforms ABD when the write ratio is below 20%. This is not surprising: ZAB does more work on writes and less work on reads in comparison to ABD.

Paxos: 129 **to** 23 **mreqs.** Paxos provides the strongest guarantees: writes have identical semantics to RMWs, and reads are linearizable (we use ABD reads for this experiment). Therefore, it is no surprise that Paxos has strictly lower throughput when compared to ABD. How does Paxos stack up against ZAB? ZAB and Paxos offer RMW semantics for its writes, while ZAB offers local reads. On that basis, it would be reasonable to expect ZAB to strictly outperform Paxos. However, we observe that ZAB only outperforms Paxos for write ratios lower than 50%, suggesting that Paxos writes are actually faster than ZAB writes. We confirm this in §8.2 and offer a potential explanation.

Kite: 526 **to** 84 **mreqs .** With synchronization accesses pegged at 5% (i.e. 5% of writes are releases, 5% of reads are acquires), Kite's performance is within 31% to 12% of ES. *This suggests that applications whose synchronization accesses constitute about 5% (or lesser) are able to reap the benefits of strong consistency at a performance that is close to EC.*

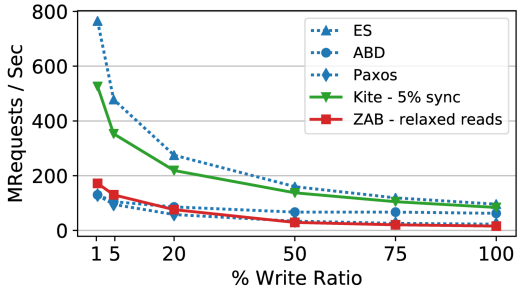


Figure 7. Throughput while varying write ratio.

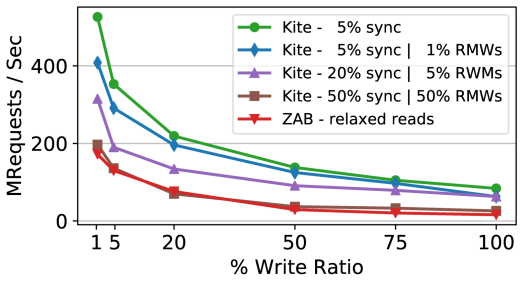


Figure 8. Kite vs ZAB while varying synchronization.

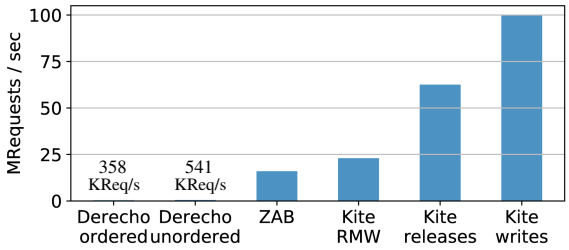


Figure 9. Write-only throughput.

Figure 8 illustrates how Kite's throughput varies with synchronization and RMWs. Workloads range from typical synchronization of 5% to the extreme of 50% synchronization and 50% RMWs. As an example, a 60% write ratio, 50% synchronization and 50% RMWs workload implies 50% RMWs, 5% writes, 5% releases, 20% reads and 20% acquires.

Unsurprisingly, Kite's performance degrades with increasing synchronization. For example, in the synchronization-heavy 20% releases-acquires and 5% RMWs workload, Kite gets about 60% to 75% of its performance with a typical 5% synchronization workload. In the limit, Kite offers similar or better performance to ZAB while offering stronger consistency (since ZAB relaxes consistency for reads).

8.2 Write-only Throughput Study

In this section we focus on a write-only workload, which not only allows us to compare against Derecho, but also allows us to derive useful insights on Kite and ZAB. Figure 9 shows

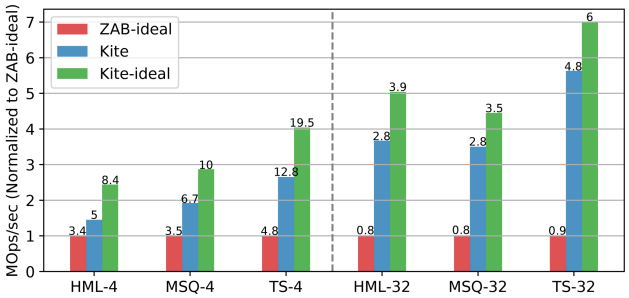


Figure 10. Normalized throughput to ZAB-ideal. Bars are tagged with flat throughput in million operations per second.

the write-only throughput (mreqs) of Derecho, Kite and our in-house ZAB. The three different types of writes of Kite correspond to Paxos (RMWs), ABD (releases) and ES (writes). We also evaluate both flavors of Derecho's atomic broadcasts: ordered and unordered.

Derecho . Derecho's comparatively low performance appears to stem from its lack of multi-threading. Utilizing high-bandwidth RDMA NICs requires multiple threads that actively send and receive messages. We believe Derecho's design focuses on huge messages (in the order of MBs), where fewer threads are required to achieve good utilization of the NICs. We note that our evaluation of Derecho is on par with recently published numbers by its authors [38].

Kite and ZAB . Kite's writes (ES writes) enjoy the highest throughput (96 mreqs) due to their lower consistency guarantees. Kite's releases (62 mreqs) offer lin (ABD writes), but still offer a lower consistency guarantee than Kite's RMWs (Paxos) and ZAB, both of which solve consensus.

ZAB vs Paxos. Paxos writes (23 mreqs) comfortably outperform ZAB writes (16 mreqs). This is because our Paxos implementation is better in uncovering request-level parallelism across RMWs to different keys. Whereas ZAB constraints parallelism by totally ordering all of the writes and applying them in the same order in all nodes, our per-key Paxos allows threads to execute RMWs on different keys in parallel.

8.3 Lock-free data structures

Using the Kite API, we implement three widely used lock-free data structures: 1) the Treiber Stack (TS) [19], 2) the Michael-Scott Queue (MSQ) [65, 66] and 3) the Harris and Michael List (HML) [31, 64]. Below, we describe how the data structures are implemented using TS as an example.

Implementation. We set up 5000 TSs, replicated across five Kite nodes. In each node there are four client threads, running 200 sessions each, issuing their requests to the workers (20 per node). Each session executes the ported TS code (from [73], including the ABA counters) as follows: it randomly picks one of the TSs and it performs a push and then a pop. Performing a push and then immediately a pop to the same TS guarantees that pops never find the stack empty and thus always incur the complete pop overhead. When multiple sessions attempt to modify a TS concurrently, their operations are said to *conflict* and must typically be retried. In order to mitigate the conflict overheads, we leverage the weak version of CAS, which can fail locally, if the compare fails locally (see § 6.1).

Correctness & Failures . We check correctness of the implementations as follows. Firstly, we assert that a pop can never find the stack empty. In addition, every object stores information about its current state in the metadata (e.g., if it is pushed and in which stack). On popping an object, we check the consistency of its metadata, with the pop action. In addition, we emulate failures by forcing Kite machines to sleep at random times for random intervals and ensure that the rest of the machines keep operating without violating correctness.
We compare Kite against two baselines:

1. ZAB-ideal. We do not yet have an API (like Kite) over ZAB and therefore we can only estimate ZAB' performance on the data structures through traces of micro-benchmarks. Because there is no way to estimate the overhead of conflicts, we instead measure ZAB with the write ratio that corresponds to each data structure, without conflicts. For instance, if performing a TS push and a TS pop results in six reads and six writes (i.e, 50% write ratio), assuming no conflicts, then the upper bound of ZAB (i.e. ZAB-ideal) in mops for TS is its throughput (in mreqs) on 50% write ratio divided by six (i.e. the number of requests required per operation).

2. Kite-ideal. In order to measure the upper bound of Kite (i.e. ideal scenario without conflicts), we grant each session its own private data structure, completely eliminating conflicts.

Figure 10 shows the performance of Kite and Kite-ideal normalized to ZAB-ideal for the three data structures. MSQ-4 is the MSQ workload, where each object has four discrete 32-byte fields, and thus requires four writes to be created and four reads to be read. Similarly, MSQ-32 is the workload where each object has 32 fields. Each bar in Figure 10 is tagged with the number of mops (million operations per second) achieved by each system. E.g. for TS, 6 mops means 3 million pushes and 3 million pops.

Comparison . Kite-ideal outperforms Kite because it does not have conflicts. Kite outperforms ZAB-ideal for all workloads from 1.45× (HML-4) to 5.62× (TS-32). The gap between Kite and ZAB-ideal is correlated with the *percentage of synchronization access required per operation* (dubbed '*sync-per*'). For instance, when the fields per object increase (from MSQ-4 to MSQ-32) the sync-per reduces, because reads and writes to the object fields are relaxed.

8.4 Failure Study

In order to study the behaviour of Kite when failures occur, we perform an experiment where a replica sleeps for 400ms.

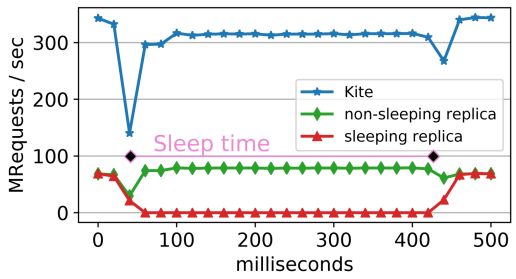


Figure 11. Failure study.

Note that, forcing a process to sleep creates a bigger challenge than simply killing it, as Kite must not only graciously handle the replica being unresponsive, but also deal with its return to normal operation, when it wakes up. Figure 11 shows the throughput over time in milliseconds (ms) of Kite in conjunction with the individual throughput of a non-sleeping and a sleeping (for 400ms) replica during the run. The workload is 5% writes and 5% synchronization. We break down the run into stable and transitioning periods. There are two transitioning periods for the sleeping replica; one that begins when its threads gradually get to sleep ($\sim$ 20ms) and another that begins when they start to wake up ($\sim$ 420ms). The stable periods are the three periods where the system throughput is steady, the pre-sleep (0-20ms), the intermediate (60-420ms) and the post-sleep (after 460ms) periods.

As expected in the pre-sleep and post-sleep periods Kite's performance is the same: 68 mreqs per machine, with a total of 342 mreqs for all 5 machines. In the intermediate stable period, we see that although the overall performance of Kite (315 mreqs) slightly drops compared to the other steady states, the throughput per (operational) node increases (78.8 mreqs) since the operational replicas are able to utilize the network resources that the sleeping replica released.

Moreover, we observe that Kite always remains available and that its transitioning periods are very small, in the range of tens of milliseconds. We also note that, although the second transitioning period involves the slow-path, it is very short since each key need only be accessed once in the slow path.

Time-out and Availability. As described in Section 4.2, when the replica sleeps, the rest of the replicas block for the duration of a time-out, waiting for the sleeping replica to ack their writes. That effect is visible on the non-sleeping replica's throughput in Figure 11. We implement the time-out with a software counter, and overprovision it ($\sim$ 1ms), such that it never gets triggered while in common operation. We note that the time-out can be arbitrarily small, but it should generally be set with respect to the system's environment.

9 Related Work

Synchronous Protocols and Systems. We refer to a system as synchronous if it assumes that failures are reliably detected. Well-known synchronous protocols include Primary-Backup [3] and Chain Replication [78, 82]. Such protocols exploit that there is no ambiguity as to whether a long delay is due to failure or not, as the system assumes perfect knowledge of which machines are alive often via an external Perfect Failure Detector (PFD). However, PFDs are known to be hard to realize in practice [52]. Kite does not rely on synchrony; it does not need an PFD.

Multiple Consistency Level Systems. There has been substantial research towards providing a multiple consistency level (MCL) API [20, 36, 54, 74, 80, 81, 84, 89] and taming them [5, 14, 29, 30, 34, 35, 53, 67, 75, 79]. While promising, we argue that merely labelling accesses (or objects) with their consistency level is not sufficient; the API should allow for expressing the ordering relationships between the strong and weak accesses. Taking inspiration from shared memory, we advocate the adoption of RC for distributed KVSs.

Causal Consistency (CC). There has been substantial work in understanding, developing and optimizing protocols to enforce CC [2, 9, 21, 22, 57, 58, 62, 63]. CC is the degenerate case of RC (but not RC_{SC}), where all writes are releases and all reads are acquires. Therefore, CC fundamentally cannot offer better performance than RC.

Software and Hardware DSMs. RDMA has sparked a recent resurgence in Software DSMs [17, 44, 68], following seminal work in the nineties [18, 46, 55, 76]. Notably, Argo [44] targets DRF programs, while TreadMarks [46], Munin [18] and Cashmere-2L [76] all implement variants of RC. Traditionally, DSMs have tended to focus on a simplistic "all or nothing" failure model [77]. Fast non-volatile memory (NVM) has renewed interest on techniques [28, 37, 40, 69, 86] that ensure the consistency of data resident in NVM upon a crash, in order to aid recovery [24]. Whereas the above systems focus on durability, considering a failure model in which all processes crash together, Kite focuses on availability, with a failure model in which individual nodes can fail in a crash-stop manner. Integrating durability is future work.

10 Conclusion

We presented Kite, the first highly-available, replicated KVS that offers a linearizable variant of RC in an asynchronous environment with crash-stop and network failures. Kite incorporates a novel fast/slow path mechanism to enforce the RC barrier semantics and is implemented in an RDMA-enabled and heavily multi-threaded manner. Kite's familiar RC API provides a pathway for the seamless porting of fault-tolerant shared memory algorithms—e.g., nonblocking data structures—for distributed KVSs. Our experimental results on three widely used lock-free data structures suggests that Kite significantly improves upon the state-of-the-art, providing a $1.5-5.6\times$ performance improvement over an in-house optimized implementation of ZAB.

References

[1] Sarita V. Adve and Mark D. Hill. 1990. Weak Ordering — a New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA '90)*. ACM, New York, NY, USA, 2–14. https://doi.org/10.1145/325164.325100

[2] 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. https://doi.org/10.1145/2465351.2465361

[3] Peter A. Alsberg and John D. Day. 1976. A Principle for Resilient Sharing of Distributed Resources. In *Proceedings of the 2Nd International Conference on Software Engineering (ICSE '76)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 562–570. http://dl.acm.org/citation.cfm?id=800253.807732

[4] 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. http://dl.acm.org/citation.cfm?id=3291656.3291659

[5] Masoud Saeida Ardekani and Douglas B. Terry. 2014. A Self-configurable Geo-replicated Cloud Storage System. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, USA, 367–381. http://dl.acm.org/citation.cfm?id=2685048.2685077

[6] ARM Limited 2018. *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile*. ARM Limited. Initial v8.4 EAC release.

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

[8] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. 2011. Laws of Order: Expensive Synchronization in Concurrent Algorithms Cannot Be Eliminated. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 487–498. https://doi.org/10.1145/1926385.1926442

[9] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Bolt-on Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. ACM, New York, NY, USA, 761–772. https://doi.org/10.1145/2463676.2465279

[10] 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/. (Accessed on 07/08/2019).

[11] Luiz André 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.

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

[13] 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 USENIX Conference on Annual Technical Conference (USENIX ATC'13)*. USENIX Association, Berkeley, CA, USA, 49–60. http://dl.acm.org/citation.cfm?id=2535461.2535468

[14] Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin Vechev. 2017. Serializability for Eventual Consistency: Criterion, Analysis, and Applications. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 458–472. https://doi.org/10.1145/3009837.3009895

[15] Sebastian Burckhardt. 2014. Principles of Eventual Consistency. *Found. Trends Program. Lang.* 1, 1-2 (Oct. 2014), 1–150. https://doi.org/10.1561/2500000011

[16] Matthew Burke, Audrey Cheng, and Wyatt Lloyd. 2020. Gryff: Unifying Consensus and Shared Registers. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 591–617. https://www.usenix.org/conference/nsdi20/presentation/burke

[17] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. 2018. Efficient Distributed Memory Management with RDMA and Caching. *Proc. VLDB Endow.* 11, 11 (July 2018), 1604–1617. https://doi.org/10.14778/3236187.3236209

[18] John B. Carter. 1995. Design of the Munin Distributed Shared Memory System. *J. Parallel Distrib. Comput.* 29, 2 (Sept. 1995), 219–227. https://doi.org/10.1006/jpdc.1995.1119

[19] Thomas J. Watson IBM Research Center and R.K. Treiber. 1986. *Systems Programming: Coping with Parallelism*. International Business Machines Incorporated, Thomas J. Watson Research Center. https://books.google.co.uk/books?id=YQg3HAAACAAJ

[20] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. 2008. *PNUTS: Yahoo!'s hosted data serving platform*. Technical Report. IN PROC. 34TH VLDB.

[21] Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. 2013. Orbe: Scalable Causal Consistency Using Dependency Matrices and Physical Clocks. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC '13)*. ACM, New York, NY, USA, Article 11, 14 pages. https://doi.org/10.1145/2523616.2523628

[22] Jiaqing Du, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. 2014. GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14)*. ACM, New York, NY, USA, Article 4, 13 pages. https://doi.org/10.1145/2670979.2670983

[23] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (April 1985), 374–382. https://doi.org/10.1145/3149.214121

[24] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. 2018. A Persistent Lock-free Queue for Non-volatile Memory. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18)*. ACM, New York, NY, USA, 28–40. https://doi.org/10.1145/3178487.3178490

[25] Vasilis Gavrielatos. [n. d.]. Kite code base. github.com/vasigavr1/kite.

[26] 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 Thirteenth EuroSys Conference (EuroSys '18)*. ACM, New York, NY, USA, Article 21, 15 pages. https://doi.org/10.1145/3190508.3190550

[27] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. 1990. Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA '90)*. ACM, New York, NY, USA, 15–26. https://doi.org/10.1145/325164.325102

[28] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. 2018. Persistency for Synchronization-free Regions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 46–61. https://doi.org/10.1145/3192366.3192367

[29] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2016. 'Cause I'm Strong Enough: Reasoning About

Consistency Choices in Distributed Systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 371–384. https://doi.org/10.1145/2837614.2837625

[30] Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. 2016. Incremental Consistency Guarantees for Replicated Objects. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 169–184. http://dl.acm.org/citation.cfm?id=3026877.3026891

[31] Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Proceedings of the 15th International Conference on Distributed Computing (DISC '01)*. Springer-Verlag, London, UK, UK, 300–314. http://dl.acm.org/citation.cfm?id=645958.676105

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

[33] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492. https://doi.org/10.1145/78969.78972

[34] Brandon Holt, James Bornholt, Irene Zhang, Dan Ports, Mark Oskin, and Luis Ceze. 2016. Disciplined Inconsistency with Consistency Types. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC '16)*. ACM, New York, NY, USA, 279–293. https://doi.org/10.1145/2987550.2987559

[35] Brandon Holt, Irene Zhang, Dan Ports, Mark Oskin, and Luis Ceze. 2015. Claret: Using Data Types for Highly Concurrent Distributed Transactions. In *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '15)*. ACM, New York, NY, USA, Article 4, 4 pages. https://doi.org/10.1145/2745947.2745951

[36] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (USENIXATC'10)*. USENIX Association, Berkeley, CA, USA, 11–11. http://dl.acm.org/citation.cfm?id=1855840.1855851

[37] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings*. 313–327. https://doi.org/10.1007/978-3-662-53426-7_23

[38] 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. *ACM Trans. Comput. Syst.* 36, 2, Article 4 (April 2019), 49 pages. https://doi.org/10.1145/3302258

[39] 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 Association, Renton, WA, 35–49. https://www.usenix.org/conference/nsdi18/presentation/jin

[40] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas. 2018. DHTM: Durable Hardware Transactional Memory. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 452–465. https://doi.org/10.1109/ISCA.2018.00045

[41] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 1–16. https://www.usenix.org/conference/nsdi19/presentation/kalia

[42] 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. http://dl.acm.org/citation.cfm?id=3026959.3027000

[43] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-sided (RDMA) Datagram RPCs. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 185–201. http://dl.acm.org/citation.cfm?id=3026877.3026892

[44] Stefanos Kaxiras, David Klaftenegger, Magnus Norgren, Alberto Ros, and Konstantinos Sagonas. 2015. Turning Centralized Coherence and Distributed Critical-Section Execution on Their Head: A New Approach for Scalable Distributed Shared Memory. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '15)*. ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/2749246.2749250

[45] Idit Keidar and Sergio Rajsbaum. 2003. On the Cost of Fault-Tolerant Consensus When There Are No Faults – A Tutorial. In *Dependable Computing*, Rogério de Lemos, Taisy Silva Weber, and João Batista Camargo (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 366–368.

[46] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. 1994. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference (WTEC'94)*. USENIX Association, Berkeley, CA, USA, 10–10. http://dl.acm.org/citation.cfm?id=1267074.1267084

[47] Christoph Lameter. 2005. Effective synchronization on Linux/NUMA systems. In *Gelato Conference*, Vol. 2005. http://www.lameter.com/gelato2005.pdf

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

[49] L. Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* C-28, 9 (Sept 1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

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

[51] Leslie Lamport. 2005. Generalized consensus and Paxos. (2005). https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2005-33.pdf

[52] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. 2011. Detecting Failures in Distributed Systems with the Falcon Spy Network. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 279–294. https://doi.org/10.1145/2043556.2043583

[53] Cheng Li, Joao Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. 2014. Automating the Choice of Consistency Levels in Replicated Systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 281–292. https://www.usenix.org/conference/atc14/technical-sessions/presentation/li_cheng_2

[54] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making Geo-replicated Systems Fast As Possible, Consistent when Necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, USA, 265–278. http://dl.acm.org/citation.cfm?id=2387880.2387906

[55] Kai Li and Paul Hudak. 1989. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. Comput. Syst.* 7, 4 (Nov. 1989), 321–359. https://doi.org/10.1145/75104.75105

[56] 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 USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*. USENIX Association,

Berkeley, CA, USA, 429–444. http://dl.acm.org/citation.cfm?id=2616448.2616488

[57] 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 Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 401–416. https://doi.org/10.1145/2043556.2043593

[58] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2013. Stronger Semantics for Low-latency Geo-replicated Storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (nsdi'13)*. USENIX Association, Berkeley, CA, USA, 313–328. http://dl.acm.org/citation.cfm?id=2482626.2482657

[59] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. 2019. A Formal Analysis of the NVIDIA PTX Memory Consistency Model. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. ACM, New York, NY, USA, 257–270. https://doi.org/10.1145/3297858.3304043

[60] Nancy A. Lynch. 1996. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[61] N. A. Lynch and A. A. Shvartsman. 1997. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*. 272–281. https://doi.org/10.1109/FTCS.1997.614100

[62] Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin. 2011. *Consistency, availability, convergence*. Technical Report. Univ. of Texas at Austin. https://www.cs.cornell.edu/lorenzo/papers/cac-tr.pdf

[63] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. 2017. I Can'T Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (NSDI'17)*. USENIX Association, Berkeley, CA, USA, 453–468. http://dl.acm.org/citation.cfm?id=3154630.3154668

[64] Maged M. Michael. 2002. High Performance Dynamic Lock-free Hash Tables and List-based Sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '02)*. ACM, New York, NY, USA, 73–82. https://doi.org/10.1145/564870.564881

[65] Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '96)*. ACM, New York, NY, USA, 267–275. https://doi.org/10.1145/248052.248106

[66] Maged M. Michael and Michael L. Scott. 1998. Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors. *J. Parallel Distrib. Comput.* 51, 1 (May 1998), 1–26. https://doi.org/10.1006/jpdc.1998.1446

[67] Matthew Milano and Andrew C. Myers. 2018. MixT: A Language for Mixing Consistency in Geodistributed Transactions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 226–241. https://doi.org/10.1145/3192366.3192375

[68] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2015. Latency-Tolerant Software Distributed Shared Memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 291–305. https://www.usenix.org/conference/atc15/technical-session/presentation/nelson

[69] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 265–276. http://dl.acm.org/citation.cfm?id=2665671.2665712

[70] 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. https://doi.org/10.1145/2749246.2749267

[71] 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, New York, NY, USA, Article 2, 6 pages. https://doi.org/10.1145/1529974.1529978

[72] Denis Rystsov. 2018. CASPaxos: Replicated State Machines without logs. *CoRR* abs/1802.07000 (2018). arXiv:1802.07000 http://arxiv.org/abs/1802.07000

[73] Michael L. Scott. 2013. *Shared-Memory Synchronization*. Morgan & Claypool Publishers.

[74] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free Replicated Data Types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems (SSS'11)*. Springer-Verlag, Berlin, Heidelberg, 386–400. http://dl.acm.org/citation.cfm?id=2050613.2050642

[75] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. 2015. Declarative Programming over Eventually Consistent Data Stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 413–424. https://doi.org/10.1145/2737924.2737981

[76] Robert Stets, Sandhya Dwarkadas, Nikolaos Hardavellas, Galen Hunt, Leonidas Kontothanassis, Srinivasan Parthasarathy, and Michael Scott. 1997. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-write Network. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP '97)*. ACM, New York, NY, USA, 170–183. https://doi.org/10.1145/268998.266675

[77] Chunqiang Tang, DeQing Chen, Sandhya Dwarkadas, and Michael L. Scott. 2003. Efficient Distributed Shared State for Heterogeneous Machine Architectures. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS '03)*. IEEE Computer Society, Washington, DC, USA, 560–. http://dl.acm.org/citation.cfm?id=850929.851916

[78] 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. http://dl.acm.org/citation.cfm?id=1855807.1855818

[79] Douglas B. Terry. 2013. Replicated data consistency explained through baseball. *Commun. ACM* 56 (2013), 82–89.

[80] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. 2013. Consistency-based Service Level Agreements for Cloud Storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 309–324. https://doi.org/10.1145/2517349.2522731

[81] 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. Ithaca, NY, USA.

[82] 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 - Volume 6 (OSDI'04)*. USENIX Association, Berkeley, CA, USA, 7–7. http://dl.acm.org/citation.cfm?id=1251254.1251261

[83] Paolo Viotti and Marko Vukolić. 2016. Consistency in Non-Transactional Distributed Storage Systems. *ACM Comput. Surv.* 49, 1, Article 19 (June 2016), 34 pages. https://doi.org/10.1145/2926965

[84] Werner Vogels. [n. d.]. Choosing Consistency, All Things Distributed. https://www.allthingsdistributed.com/2010/02/strong_consistency_simpledb.html. (Accessed on 11/04/2019).

[85] Werner Vogels. 2009. Eventually Consistent. *Commun. ACM* 52, 1 (Jan. 2009), 40–44. https://doi.org/10.1145/1435417.1435432

[86] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 91–104. https://doi.org/10.1145/1950365.1950379

[87] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. 2017. APUS: Fast and Scalable Paxos on RDMA. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*. ACM, New York, NY, USA, 94–107. https://doi.org/10.1145/3127479.3128609

[88] Andrew Waterman, Yunsup Lee, David A. Patterson, Krste Asanovic, Volume I User level Isa, Andrew Waterman, Yunsup Lee, and David Patterson. 2014. The RISC-V Instruction Set Manual.

[89] Alex Yarmula. 2016. Strong consistency in Manhattan. https://bit.ly/36rRVLj. (Accessed on 10/12/2019).