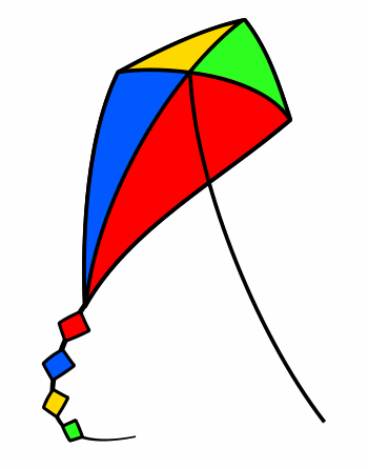
Kite: Efficient and Available Release Consistency for the Datacenter

Supplementary Material



Contents

1 What is the purpose of this document?

This document contains material that supplements our submitted paper on Kite. Kite is a replicated Key-Value Store that offers highly available RC_{Lin}. By its very nature, any system such as Kite will include a large number of design decisions at the protocol- and implementation-level. As such it is impossible to include all of this in a 10-page paper, while also motivating and explaining all of our decisions. Instead, we choose to provide as complete a specification of Kite as possible in the paper submission, while using this document to provide clarification for questions that may arise.
Below we briefly enumerate the contents of this document:
Proof of fast/slow path (§ 2). The most important part of this document is the informal, but rigorous, proof of Kite's fast/slow path mechanism that is sketched in this section.
System implementation (§ 3) This is a super-set of the corresponding system section on the submitted paper. Here, we unravel some more details of Kite's implementation, providing details about Kite's API and how we tested Kite.
Frequently Asked Questions (§ 4) In this section, we attempt to identify and answer questions around some of the more controversial design decisions of Kite. We include answers to issues such as: Why leaderless Paxos? Why use ABD and not always Paxos? and more.

2 Proof sketch for Kite's Fast/Slow Path

Kite employs two mechanisms to enforce the barrier semantics of Release Consistency (RC) : 1) the *fast-path* which is very efficient, but blocking and hence cannot operate on an asynchronous environment and 2) the nonblocking *slow-path* which can operate on an asynchronous environment with machine crashes and network failures, but incurs a considerable overhead. Kite alternates among the two paths: it leverages the fast-path for performance but hinges on the slow-path for progress when asynchrony presents. In this section, we first discuss the mechanisms, then formalize RC and finally we prove informally, but rigorously, that the fast-path/slow-path mechanism enforces RC.

2.1 Kite's fast/slow path.

Below we provide a summary of Kite's fast-path/slow-path mechanism for completeness.
Release. Before a release can execute, it attempts to gather acks (from all replicas) for each prior write in session order.

– Fast-path release. If all prior writes have been acked by all replicas, the release simply executes (by performing an ABD write).

– Slow-path release. If any preceding write has not been acked by all replicas within a time-out, the slow-path release is executed as follows. Before the release begins executing it enforces two invariants: (1) all previous writes have been acked

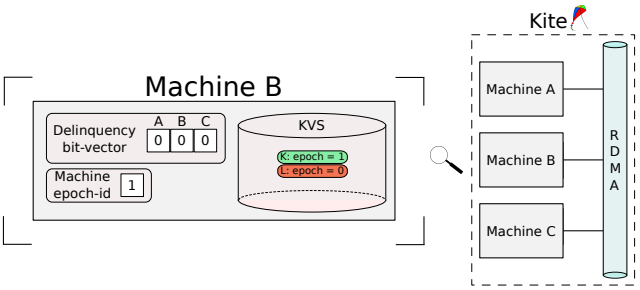


Figure 1: Zooming inside Machine B. Key L is ouf-of-epoch (slow-path); key K is in-epoch (fast-path).

by at least a quorum of machines and (2) the set of delinquent machines (i.e the set of machines that have not acked one or more of the writes) is known to at least a quorum of machines. To satisfy (2), a *slow-release* message is broadcast, containing the identities of the delinquent machines. The release begins executing only after a quorum of machines have acknowledged 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). Note that if the acquire requires a second round, then replies to the second round must also notify the acquirer if it has been deemed delinquent.

– 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-wide *epoch-id*, rendering the locally stored keys *out-of-epoch* (we further describe the semantics of epochs below).

Epochs. As shown in Figure 1, 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 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). Conversely, 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).

Returning to fast path. The transition to the fast path happens at a per-key granularity. Accessing a key in the slow-path guarantees that the key is up-to-date. Therefore, upon accessing an out-of-epoch key, the key's epoch-id is advanced to the machine's epoch-id, bringing the key back in-epoch. Note that the key's epoch-id is advanced to what the machine epoch-id was when the relaxed accessed started, rather than to the value of the machine epoch-id when the relaxed access completes.

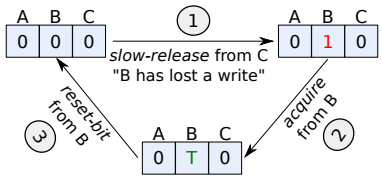


Figure 2: The transitions of the bit-vector of machine A, in a configuration with 3 machines: A, B & C

2.1.1 Delinquency Bit Vectors

In a Kite deployment, each machine maintains a bit-vector with a delinquency bit for each remote machine. Delinquency bits get set upon receiving a slow-release message. On receiving a slow-release message, delinquency bits are unconditionally set (i.e. to 1) for all machines that are in the delinquency set of the slow-release message. On receiving an acquire, the receiver notifies the acquirer about their delinquency if the corresponding delinquency bit is set. In addition, the receiver transitions the bit to a transient state T, and keeps note of the acquirer's id (i.e. the id of the acquire message) to allow the acquire to reset the bit in the future. If the receiver receives more acquires from the same machine (but different sessions) then it also keeps note of their ids, keeping the bit in state T. The acquirer, after having been notified of its delinquency and incremented its machine epoch-id, broadcasts a *reset-bit* message that is used to reset the delinquency bits of remote machines. On receiving a reset-bit message, the delinquency bit of the sender can be reset iff 1) it is in state T and 2) the reset-bit carries the same id as one of the acquires that are noted for that bit. Note that receiving a slow-release message unconditionally sets relevant bits to 1.

Example. For instance, consider a deployment with machines A, B and C. Figure 2 illustrates the transitions of A's bit-vector. When A receives a *slow-release* message from C denoting that B is delinquent, A sets the bit for B in its bit-vector. When B sends an acquire message to A, it is informed of its delinquency and transitions to the slow path. A transitions its bit to a transient (T) state, upon receiving the acquire from B. A transitions the bit back to 0, iff the bit is in T state and it receives a *reset-bit* message spawned from the same acquire that transitioned the bit to T state. To correlate *reset-bit* messages and acquires, each acquire is tagged with a unique id, which is included in any generated *reset-bit* message. A simply stores the ids of all of B's acquires since receiving the last *slow-release* message containing B. The required storage for the acquire ids is bounded by B's sessions, as B may only have one outstanding acquire per session.

2.2 Release Consistency Semantics

We use the following notation for memory events:

- $\mathbf{M^i_x}$: a memory operation (of 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. $M1^i_x$)

- $\mathbf{Rel_x^i}$: a release (release write or release-RMW) to key x from session i.
- $\mathbf{Acq_x^i}$: 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:

- $\mathbf{M_x^i} \xrightarrow{\text{so}} \mathbf{M_y^i}$: $\mathbf{M_x^i}$ **precedes** $\mathbf{M_y^i}$ in session order.
- $\mathbf{M_x^i} \xrightarrow{\text{hb}} \mathbf{M_y^j}$: $\mathbf{M_x^i}$ **precedes** $\mathbf{M_y^j}$ in the global history of memory events, which we refer to as happens-before order ($\xrightarrow{\text{hb}}$).

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: $M_x^i \xrightarrow{\text{so}} Rel_y^i \Rightarrow M_x^i \xrightarrow{\text{hb}} Rel_y^i$.

ii) A memory access that follows an acquire in session order appears after the acquire in happens-before: $Acq_y^i \xrightarrow{\text{so}} M_x^i \Rightarrow Acq_y^i \xrightarrow{\text{hb}} M_x^i$.

iii) An acquire that follows a release in session order appears before the release in happens-before: $Rel_y^i \xrightarrow{\text{so}} Acq_x^i \Rightarrow Rel_y^i \xrightarrow{\text{hb}} Acq_x^i$.

iv) Two memory accesses to the same key ordered in session order preserve their ordering in happens-before: $M1_x^i \xrightarrow{\text{so}} M2_x^i \Rightarrow M1_x^i \xrightarrow{\text{hb}} M2_x^i$.

v) RMW-atomicity axiom: an RMW appears to executes atomically, i.e. for an RMW that is composed of a read R_x^i and a write W_x^i, there can be no write W_x^j such that $R_x^i \xrightarrow{\text{hb}} W_x^j \xrightarrow{\text{hb}} W_x^i$.

vi) Load value axiom: A read to a key always reads the latest write to that key before the read in happens-before: if $W_x^j \xrightarrow{\text{hb}} R_x^i$ (and there is no other intervening write W_x^k such that $W_x^j \xrightarrow{\text{hb}} W_x^k \xrightarrow{\text{hb}} R_x^i$), the read R_x^i reads the value written by the write W_x^j.

2.3 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, along the way identifying the non-trivial cases that need to be proved more rigorously.

Recall that in a well-formed program (aka data-race-free program) synchronization variables are correctly marked using releases/acquires. If the read in the load value axiom is marked acquire and the write is marked release, the ABD protocol actions (that enforce linearizability) automatically enforce the load value axiom.

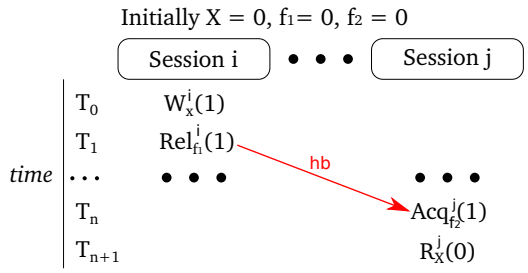


Figure 3: Proof sketch assumed violation. The violations is that Session A reads $X = init$, while RC mandates that it must read $X = 1$.

Another 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 for each session. More specifically, the previous write either would been applied to the KVS already (fast path) or will be bypassed (slow path).

Therefore, the interesting case is when the write and read are relaxed operations from two different sessions: specifically W_x^i (from session-i) and R_x^j (from session-j). 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 3, given that $W_x^i \xrightarrow{\text{so}} Rel_{f_1}^i \xrightarrow{\text{hb}} Acq_{f_2}^j \xrightarrow{\text{so}} R_x^j$, we need to prove that R_x^j will read the value written by W_x^i (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.

Lemma 2.1. *$Acq_{f_2}^j$ cannot complete its execution (in real time) before $Rel_{f_1}^i$ begins execution.*

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

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

Let us assume that both session-i and session-j are operating in fast-path[1]. In such a case, Kite ensures the load value axiom via the following real-time orderings:

- Kite waits for all writes before a release to be acked before performing the release. This means that W_x^i completes before $Rel_{f_1}^i$ begins.
- $Acq_{f_2}^j$ cannot complete its execution (in real time) before $Rel_{f_1}^i$ begins execution. (from Lemma 2.1)

[1]More precisely, the machine(s) in which the sessions are mapped to are operating in the fast-path

- Kite blocks the acquiring session until the acquire completes. This means that R_x^j will begin execution only after the acquire $Acq_{f_2}^j$ completes.

The above real-time orderings implies that R_x^j will begin execution only after W_x^i completes, and hence will read the correct value.

2.3.2 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^i, owing to a failure (e.g. message delays or lost message). In this case, the read R_x^j must still return the value written by the write, and not return the stale value by reading 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 2.2. *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 set of delinquent machines (i.e. 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 2.3. *For a release $Rel_{f_1}^i$ and an acquire $Acq_{f_2}^j$, with $i \neq j$, and $Rel_{f_1}^i \xrightarrow{hb} Acq_{f_2}^j$, and if $Rel_{f_1}^i$ happens to publish delinquent machines before its execution, then $Acq_{f_2}^j$ 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 set of delinquent machines published by the release (dubbed 'DM-set') must have already reached a quorum of replicas. Therefore, it follows that *if the released value can be seen, the DM-set must have reached a quorum of replicas*. This is the *release invariant*.

Case a: the acquire synchronizes with the release. That is, the acquire $Acq_{f_2}^j$ reads the value of release $Rel_{f_1}^i$. (This is only possible if $f_1 = f_2 = f$). Following ABD, an acquire gathers responses from a quorum of replicas, and reads the most recent value (the value with the highest Logical Lamport Clock). 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^j reads the value of Rel_f^i 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^j is guaranteed to see

the DM-set in the intersection replica. 2) if Acq_f^j reads the value of Rel_f^i from fewer than a quorum of machines, then Acq_f^j will include a second broadcast round to write the value. In that case, it is guaranteed that the second broadcast round of Acq_f^j will begin only after the value of Rel_f^i 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^j must intersect with the quorum of machines that have seen the DM-set.

Case b: acquire does not synchronize with the release. I.e $Acq_{f_2}^j$ does not read from $Rel_{f_1}^i$. However, the fact that $Rel_{f_1}^i \xrightarrow{\text{hb}} Acq_{f_2}^j$, means that $Acq_{f_2}^j$ is at the end of a synchronization chain of releases and acquires and $Rel_{f_1}^i$ is at the top of that chain; that chain must include a release/acquire that saw the value written by $Rel_{f_1}^i$, 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 $Acq_{f_2}^j$. Therefore it follows that by the time the value f_2 can be read, the DM-set would have 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 2.4. *If an $Acq_{f_2}^j$ 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 $Acq_{f_2}^j$ discovers it has been deemed delinquent and thus it increments the machine's epoch-id (in order to transition to the slow path), before completing the acquire at time T_1. Therefore, it follows that at time T_1, the machine's epoch-id, will be 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 $Acq_{f_2}^j$, 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 three invariants, we are now in a position to prove the load-value axiom.

Lemma 2.5. *For a write W_x^i, release $Rel_{f_1}^i$, acquire $Acq_{f_2}^j$ and a read R_x^j such that: $W_x^i \xrightarrow{so} Rel_{f_1}^i \xrightarrow{hb} Acq_{f_2}^j \xrightarrow{so} R_x^j$, and if there is no intervening write to X between W_x^i and R_x^j, R_x^j will read the value written by W_x^i.*

Proof. First, we observe that $Acq_{f_2}^j$ cannot complete execution before $Rel_{f_1}^i$ begins execution. (from Lemma 2.1).

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

If R_x^j executes in the fast path, then it must be that W_x^i gathered an ack from the machine that R_x^j executes from. On the other hand, if W_x^i could not gather an ack from the machine that R_x^j executes from, then from Lemmas 2.2, 2.3, 2.4, it follows that $Rel_{f_1}^i$ will have detected accurately the DM-set and $Acq_{f_2}^j$ will have discovered its delinquency transitioning into the slow path and thus the R_x^j would happen in the slow path and hence guaranteed to read the value of W_x^i. □

2.3.3 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 be read locally in fast-path. This is safe in RC since RC only mandates 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.

2.3.4 Resetting Delinquency bits

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 this action is benign, i.e. the acquirer should never reset a bit in a manner that can cause a consistency violation. We identify two invariants that must be enforced for safety and then proceed to prove that the invariants are indeed satisfied.

First, a delinquency bit for a machine can be reset only after the machine has transitioned into its 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 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 can infer the following two lemmas that must be enforced by Kite.

Lemma 2.6. *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 2.7. *A delinquency bit that was observed by acquire Acq_x^i will be reset iff there has been no attempt to set the bit (by a racing slow-release) in between receiving Acq_x^i and its spawned reset-bit message.*

Proof. This is again enforced by Kite's actions. Each acquire is tagged with a unique monotonically increasing id, when an acquire reads a set delinquency bit it transitions that bit to a transient state T (neither set nor reset) and it writes its unique-id along with that bit. Additionally, reset-bit messages include the unique-ids of their parents. When a reset-bit message is received, it is allowed to reset the delinquency bit only if the bit is set in state T and the carried unique-id is written along with the bit. On resetting the bits, 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 seen 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 all the written 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.*

2.4 Correctness of fast/slow path optimizations

In the submitted paper we discuss two optimizations that we have implemented for the fast/slow path mechanism: 1) overlapping a release with waiting for acks and 2) optimizing the relaxed accesses of the slow-path. Below we provide an argument which shows that implemented optimizations do not interfere with our above proof, thereby preserving the correctness of Kite.

2.4.1 Overlapping a release with waiting for acks

The *release invariant* which we leverage in our proof is that: the released value must not be applied to any replica before the DM-set has reached a quorum of machines. So far, we have argued that we enforce this invariant by waiting for the DM-set to be acked by a quorum, before beginning executing the release. The proposed optimization carefully relaxes this: we wait for the DM-set to be acked by a quorum before beginning executing the *second broadcast round of the release*. This modification does not violate the original invariant because the first broadcast round of an ABD write merely reads the Lamport Logical Clocks (LLCs) of all remote replicas without applying the new value to any replica.

RMWs. Extending this optimization to RMWs, we overlap the Paxos first phase (i.e. the "propose" phase), with waiting for acks for previous writes. The propose phase of Paxos is similar to the first phase of an ABD write, in that it does not broadcast the new value, and thus it maintains the invariant that the released value cannot be applied to any replica before the DM-set has reached a quorum of machines.

2.4.2 Slow-path optimization.

Relaxed access that execute in the slow path do not execute the full ABD algorithm.
Relaxed slow-path reads. Reads execute one broadcast round (same as ABD) but never a second round, whereas ABD includes a conditional second round that broadcasts the value read. The optimization is correct, because the invariant that we require out of relaxed slow-path reads is that they reach a quorum of machines such that they can intersect with "released" writes of remote machines (relaxed writes which precede a completed release). The ABD-read invariant that we discard is that before a read completes it makes sure that its returned value has been seen by a quorum of machines. Whereas this invariant is necessary for a read to be linearizable, it is not necessary for a relaxed read.
Relaxed slow-path writes. Writes execute the two ABD rounds but do not wait for acks to the second round to complete. The invariant we want to ensure is that the relaxed slow-path write will observe all "released" writes (relaxed writes which precede a completed release), so that it can use a larger LLC than them (and serialize after them). This invariant is enforced by reading LLCs from a quorum of machines, guaranteeing an intersection with all released writes. The invariant that we discard is that a write does not return before it has reached a quorum of replicas. A relaxed write need not ensure that it has reached a quorum of replicas, because the next release in session order is tasked with enforcing that.

3 System Design

In this section, we describe how Kite is implemented and validated. First, we provide a functional overview of Kite (§ 3.1), then we provide a detailed description of Kite's API and its implementation (§ 3.2). Subsequently, we provide details on Kite's KVS (§ 3.3) and its networking (§ 3.4). Finally, we share some of our experience in validating Kite (§ 3.5).

3.1 Functional Overview

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,

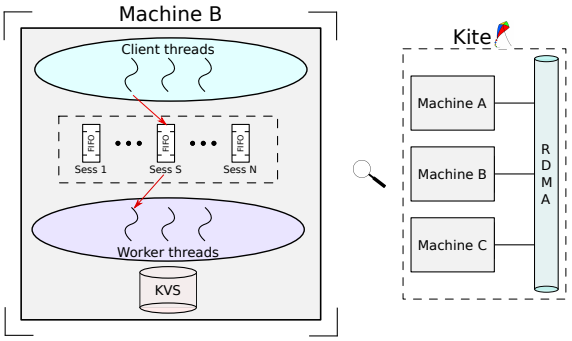


Figure 4: A Kite machine is composed of worker and client threads, that interface through the session FIFOs.

propagating the requests to the worker threads. For the rest of the paper we assume that client 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, honouring 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 necesary protocol-level messages to execute requests.

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 the thread-level parallelism across workers threads (which only need to synchronize when accessing the same key); session-level parallelism within a worker thread (as every worker is typically responsible for multiple sessions); and finally, RC allows for parallelism within one session (as the execution of relaxed accesses can be overlapped).

3.2 Kite API: the client-worker interface.

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*.

Kite 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 that can

complete locally if the comparison fails locally, and a strong that always incurs a Paxos command even if the comparison fails locally. The Kite API includes an asynchronous (*async*) and a synchronous (*sync*) function call for every request (similarly to Zookeeper [7]).

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—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; however we also plan to add support for callbacks, which are the most popular primitive for asynchronous programming.

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. A programmer 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. Other use-cases of using multiple sessions per client thread, include connecting multiple remote clients to one client thread and using Kite as an eventually consistent store.

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 5. The client fills the fields of the slot to issue a request, and the worker uses the fields to complete the

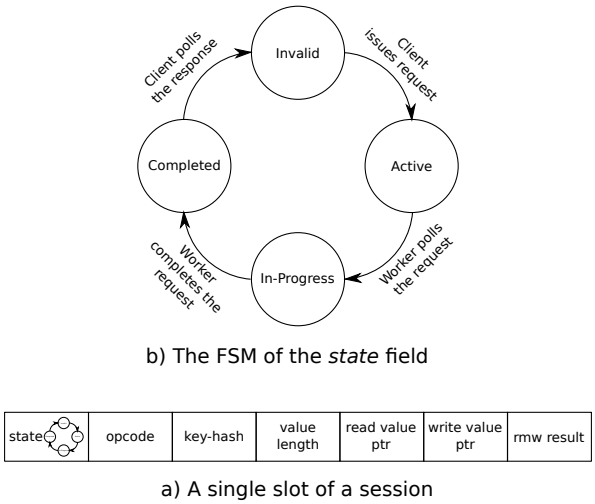


Figure 5: The fields of one slot of one session FIFO, and the FSM of the state field.

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.

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 5), 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. Worker transitions the slot to *In-progress* when it polls it and later to *Completed* when it completes it.

3.3 Key-Value Store implementation

Every node in Kite maintains a local KVS. The implementation of the KVS is largely based on MICA [15] as found in [9], with the addition of sequence locks (seqlocks) [11], from [5], to enable multi-threading.

Adapting MICA for ES and ABD. The MICA read/write API largely fits our needs for ES and ABD. Still, we make several modifications to accommodate Kite-specific actions, such as reading only the LLC of a key (for ABD writes) or adding per-key epoch-ids to enable slow/fast-path transitions.

Adapting MICA for Paxos. The MICA API cannot capture Paxos actions (e.g. proposes/accepts). We rectify that by adding a level of indirection: each MICA key contains a pointer to its own Paxos-structure. Locking the key through its seqlock also locks the respective Paxos-structure. The Paxos-structure stores necessary metadata to perform Paxos such as: highest proposed LLC, highest accepted LLC etc. Therefore, a Paxos-related request goes through MICA, locks the corresponding key and gets directed to the key's Paxos-structure, where it can act on the request.

3.4 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 [9, 10, 5, 8].

RDMA Optimizations. We carefully implement low-level, well-established RDMA practices, such as doorbell batching and inlining, (reader is referred to [2, 9, 10, 5] 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 protocol contain broadcast primitives. We implement broadcasts through unicasts in the same manner as [5].

3.5 Validating Kite

We employ three types of testing to validate Kite. Note that in all tests, we force machines/threads to sleep at random times for random intervals to simulate failures and asynchrony.

1. Asserting invariants. We identify the high-level invariants of Kite's operation (e.g., "an acquire cannot execute unless all preceding releases have completed") and we break them down into small invariants that are checked with assertions. We deeply integrate thousands of such assertions in Kite's implementation and provide a compiler option to enable them.

2. Validating Paxos logs. In order to check Paxos-related invariants (e.g. "an RMW must be executed exactly once"), we build a tool that gathers logs from all Kite machines and checks for several such RMW invariants.

3. Checking the values. Even when all invariants are correctly identified and enforced, the returned values can still be incorrect due to bugs in the data path. We check the returned values with litmus tests, including our implementations of various lock-free data structures.

4 Frequently Asked Questions (FAQ)

This section identifies and answers the most frequently asked questions (FAQs) about Kite. We group questions into subsections according to their subject. We begin with answering questions about the Kite mappings, focusing on ABD and Paxos. Then we answer questions about the fast/slow path mechanism, the system implementation and finally we focus on some general questions about Kite.

4.1 Kite mappings

4.1.1 What about Multi-Paxos? (i.e. Paxos with a leader)

Lamport proposes that when Paxos is executed repeatedly (i.e. multi-decree Paxos), it should elect a stable leader [12]. The stable leader can execute the propose phase for only the first command it commits, and avoid it for all the rest. However, introducing a leader has a number of downsides: 1) an availability penalty is incurred when the leader is unavailable [20], 2) when the leader eventually fails, a protocol should be in place to handle the transition to a new leader, 3) having a stable leader creates a network and computation imbalance in the system, limiting the throughput to the network bandwidth and computation power of one node, 4) if commands can originate from follower machines (which would be the case in Kite), they would have to be shipped to the leader, offsetting the benefits of avoiding the propose phase. Finally, we note that since we execute Paxos per-key, we could avoid the imbalance caused by a single leader by electing one leader per key. However, that approach would radically increase the complexity of handling leader failures.

Therefore, Paxos in Kite is implemented leaderless: the basic Paxos protocol is performed for every RMW. In doing that, 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, the concurrent and decentralized nature of the protocol and the correctness guarantees. Additionally, a fully distributed Paxos allows seamless integration of the RC barrier semantics with RMWs.

4.1.2 Why do we use ABD for acquire/release operations? What about Paxos with a leader? Would they not have the same number of rtts?

ABD and Paxos provide different guarantees. First let us set the stage by clarifying why Paxos and ABD are different: Releases are *full writes*, which need lower guarantees than what Paxos provides. Putting it plainly, RMWs for a given key must be executed in lock-step because any given RMW must know the value on which it operates; this is the guarantee provided by Paxos. The same is however not true for full-writes which overwrite the entire old value, and thus need not know what it was. This is precisely the reason ABD works for full writes but not for RMWs. To summarize, we use ABD because: (1) ABD exploits the nature of full writes and thus is fundamentally more efficient than Paxos and its variants (which perform consensus); and (2) ABD provides liveness guarantees that are impossible for Paxos (Paxos is obstruction-free but can have livelocks). Putting it simply, Paxos is a necessary evil for RMWs, but can be avoided for full writes.

Paxos with a leader. One may think that a multi-Paxos variant (Paxos with leader) would be as efficient or more efficient than ABD by comparing the number of rtts. Firstly, in a multi-Paxos variant, the request must find its way to leader and then the leader needs to perform two rounds of broadcast (same as an ABD write). The difference is that Multi-Paxos can report completeness after the first rtt. But there are several problems. Firstly, having a single leader generates an availability problem, as its failure must be detected on the critical path and a protocol must be in place to decide the transition to a new leader. Secondly, requests must be steered to the leader, which

means that either all clients are all connected to a single machine (i.e. the leader), or that collocated clients (collocated with any Kite machine) must incur an extra hop. Finally, having a leader sacrifices the load imbalance of the system, limiting the system's throughput to the capabilities of a single machine. (One another client problem with leaders is discussed in a separate question in § 4.1.4).

Comparing based on rtts. More importantly—and contrary to conventional wisdom—merely comparing number of RTTs is a wildly insufficient metric to gauge throughput or latency: it says nothing about load balancing, stalling/blocking, concurrency amongst independent requests, the size of the messages, or even the ability to maintain a non-blocking API in the client side. For instance, consider multi-Paxos : for every executed write the leader must do a propose broadcast (send N messages), receive acks (receive N acks) and then broadcast commits (send N messages). Compare that with a follower, which for each committed write, need only receive 2 messages and send back acks. Consequently, the network (assuming the typical star topology) and the processors are imbalanced and thus heavily underutilized (i.e. followers only use a small fraction of their network bandwidth when leader uses all of its bandwidth). That is without even considering how the writes found their way to the leader in the first place. Comparing this with ABD writes: the rtts may be the same, but there is no availability sacrifice, no extra hidden protocols for leader transitions, the resources of the system are well-balanced and writes need not be steered to the leader.

4.1.3 What about multiple leaders for different sets of keys?

Although this is a plausible design, it has several drawbacks. Firstly, any node failing will result into an availability penalty, as requests for its set of owned keys are stalled. Additionally, keys must be routed to the correct leader incurring some sort of hashing and extra hops when clients are collocated. But the truly interesting part is this: what must happen when one node fails: the failed node must be a leader for a bunch of keys and these keys must now be distributed to new leaders; what if the leader that was perceived to have failed was just unavailable for a few milliseconds and returns (an acceptable scenario in Kite); how are keys going to be distributed back to it? who is going to let the clients know about the new hashing? There needs to be a very complicated protocol running underneath managing all of this complexity. Having such a protocol in a synchronous setting may be possible, but in an asynchronous setting, where arbitrary delays in networking and processing are present, this complexity is likely unmanageable.

4.1.4 How does having a leader affect the client API?

The non-blocking API of Kite would lose a lot of its power if a client had to communicate with a leader directly: if subsequent requests from the same client are served in different Kite machines, then the client cannot issue a group of them (as it can in Kite), but it must block until the last request is completed. Having the client working in lock-step would be bad for performance, which is part of the reason why Zookeeper serves all of the requests of a client from the same machine, irrespective of whether that machine is the leader.

4.1.5 Why Basic-Paxos? What about all of its proposed variants?

We have looked into proposals that attempt to optimize Paxos. We have found that more often than not the performance benefits (if any) come with strings attached. We already discussed the (often hidden) implications of having a stable leader in questions 4.1.1, 4.1.4 and 4.1.3. A group of variants (e.g. [3, 18, 21]) use predetermined order (sometimes called rotating leaders), where each slot of the Paxos log is determinsitically given to a machine, irrespective of whether the machine has the intention to perform a Paxos command. Problems of this approach include: 1) the failure of any machine will incur an availability penalty; and 2) those machines that do not have a Paxos command to commit must explicitly let the rest of machines know in a timely manner. Epaxos [20] is based on decoupling the execution of a command from its commitment, and therefore returns to client without having executed the command, which would make RMWs hard if not impossible, because they must return the read value. Finally, other optimizations such as broadcasting acks to reduce rtts or having the clients broadcast the command to all machines are often impractical and contain hidden bottlenecks (e.g., broadcasting acks reduces rtts at the cost of network bandwidth).

All-aboard Paxos [6]: an optimization, not a trade-off. According to Lamport [13], Paxos is unavoidably derived from the properties it intends to satisfy. We agree with this sentiment. However, a natural optimization that Lamport never disproved is a fast/slow path based on the presence of asynchrony (similar to Kite fast-slow path). A natural question to ask is, can we have a fast-path Paxos that takes advantage of the common case (where all machines are alive and reply on a timely manner), while reverting to the original Paxos, when asynchrony presents itself (where Paxos is truly unavoidable). The only proposal, that we know of, that achieves this effect is All-aboard Paxos, which in the common case reduces the rtts of basic Paxos from 3 to 2. We plan to implement all-aboard Paxos in Kite and showcase its performance in the future.

4.1.6 Does per-key Paxos imply a separate log per key?

One may expect that per-key Paxos implies the need to keep a log for each key. However, KVSes that implement Paxos need not keep a log at all; rather, remembering the last committed Paxos command for each key is sufficient. In fact, Lamport discusses this effect in his original Paxos paper [12], proposing the use of a hashtable (called "law book" in his analogy) to avoid the maintenance of long logs (i.e. "long ledgers"). This observation allows Kite to avoid the storage and computation overheads of maintaining and indexing a log.

4.2 Fast/Slow Path mechanism

4.2.1 Why can we not broadcast the set of delinquent machines with the first round of a release?

The first round of a release reads the key's LLC from a quorum of machines. This first round can execute before gathering the acks for previous writes and thus can be over-

lapped with waiting for acks. If we were to piggy-back the set of delinquent machines in that first round, before issuing that first round we would have to wait for all acks to be gathered, making it impossible to overlap the round with the waiting. Therefore, this would not be a favourable trade-off.

4.2.2 Why can we not broadcast the set of delinquent machines with the second round of a release?

Because the set of delinquent machines must reach a quorum before the value is visible anywhere. To see why, suppose we did piggyback the delinquent set with the value and consider this example: machine A releases and along with the value propagates the information that B is delinquent. Then machine C acquires, sees A's released value and because the value has not reached a quorum, C takes it upon itself to perform a second broadcast round with the value. Now the value can reach a quorum but the information that B is delinquent has not. To resolve this, acquires should be able to also "help" delinquency bits, not only values. However, that would create a whole other kind of complexity—especially when trying to reset remote delinquency bits—as now the delinquency bits are not broadcast once, but potentially can be broadcast perpetually by acquires that simply try to help.

4.2.3 Why does the second round of an acquire need to also discover delinquency?

Because broadcasts are not atomic. Consider this case with 5 machines A,B,C,D,E: Machine A acquires and reaches itself and B; it must reach one more machine to reach a quorum but lets assume that all other messages it has sent are delayed. Then, E tries to release, but it misses an ack from A, and thus E broadcasts that A is delinquent and it reaches B, C and itself (E). Then, E moves on to broadcast the released value reaching B, D and itself. Then, the acquire that A had issued also reaches D, but D has seen the released value but not the fact that A is delinquent. And thus, A has now read the released value without seeing its delinquency. But A is also forced to do one more round for its acquire because the released value has not been seen by a quorum, now since A saw the released value, the second round of the acquire is guaranteed to intersect with the quorum B, C, E that now that A is delinquent.

4.2.4 The mechanism is too coarse. Can we not detect failures in a more fine-grain manner? What about vector clocks?

If a machine is slow to ack a write, it will have to refresh its entire KVS. That indeed seems very costly. However, it is the price we have to pay for realizing two key requirements: the failure model and the efficiency of common-case execution.
Common-case. Note that this mechanism rids the common case of overheads found in common Causal Consistency protocols, such as carrying bit vectors, or waiting to receive a write such that writes are applied in some specific order. Rather, Kite's common case truly proceeds as if failures do not exist.

Can we track the lost writes? Still, it is fair to wonder: if a machine loses a message, why can it not just ask for it? Why should it have to refresh its entire store? Well that question has an implicit assumption that all other machines can keep track of all the writes that a given machine has missed. We note several problems with this approach: 1) in the common case where asynchrony is due to a failure, we will be burdening all alive nodes with unnecessary logging 2) there is an implicit assumption that a machine that has lost a write can contact the writer and ask for the write, but it may be that just a link between two machines is lost, in which case how is a machine to know which writes it has missed? 3) the logging itself must be bounded somehow in which case the reasonable approach is to log the first few thousand writes and beyond that simply notify the machine to refresh its whole store. In a very related note, recall that Kite can do 93 million writes per second, which would be quite burdensome to log. These reasons are why systems typically revert to vector clocks for this problem.

Vector Clocks. CC protocols [16, 19, 17, 4] and ZAB [22] take a different approach: they enforce a single order in which writes can be applied. As such, a read can always observe a consistent state (even though potentially a stale one). To achieve that effect each write is tagged with a vector clock that denotes its dependencies (i.e., all the previous writes that must have been applied, for this write to be applied). Note that the write cannot be applied until its dependencies are applied too. The impact of this approach (other than the overhead of transferring dependencies) is twofold: 1) applying writes in a specific order implicitly assumes that a replica never fails to receive a write; if a write message is missed, the replica is not able to apply any subsequent writes, and must buffer them (possibly indefinitely); 2) enforcing an explicit order amongst writes to different keys limits thread-level parallelism, because threads are prohibited from writing the datastore in parallel even if they are writing to different keys. (Even though OCCULT [19] and ZAB do not use vector clocks, they still fall into the same category as they just use a single total order — per shard — which allows the dependencies to be expressed with just an id, instead of a vector clock.)

4.2.5 Causal Consistency seems relevant, why not compare against it?

For the following reasons: 1) Causal Consistency (CC) is the degenerate case of RC (but not RC_{SC}) where all reads and writes are tagged releases and acquires, and as such it cannot be better than RC. 2) CC protocols leverage vector clocks and as such they burden the common case with carrying dependencies on every write, while severely limiting concurrency (a write cannot be applied before its dependencies are applied) and also relaxing the failure model as now messages can never be lost. 3) CC protocols lacks the primitives to implement basic synchronization primitives (it can neither enforce ARAW nor the RAW orderings, one of which must be present to implement mutual exclusion [1]), 4) We do compare against ZAB which implements a similar approach to vector clocks, but can perform synchronization, 5) We do not know of any high-performance, RDMA-enabled systems that enforce CC (recall that even for ZAB we built our own in-house version).

4.2.6 Why can we not reset a delinquent bit right then and there when the acquire reads it?

If the acquire were to reset the bit when reading it, without having to issue a reset-bit message, then a second 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, before the first acquire had time to increment the epoch-id.

4.2.7 If resetting the delinquent bit is an atomic action then why does it not require consensus?

The key thing to realize here is that delinquent bits are not replicated. Rather each machine has its own delinquent bit vector that need not reflect the same state as the bit vectors of other machines. Therefore, an acquire does not attempt to reset the remote delinquent bits of all machines atomically, rather it makes a best-effort attempt to flip each of the remote bits individually (but atomically). If 3 out of 5 machines believe that machine A is delinquent, an A acquires and manages to reset only the delinquent of one machine, that is perfectly acceptable. Therefore, the machines need not reach consensus among them as they do not really share state. Rather, the delinquent machines sends RPCs to the remote machines, asking them to reset the bit if certain conditions are met. That action certainly does not require consensus.

4.2.8 How can a delinquency bit store all the unique id of the acquires that attempt to reset it? Wouldn't that require unbounded storage?

Because a session can only have on outstanding acquire, the maximum number of outstanding acquires a machine can have is bounded by the number of sessions per machine. That is how many slots we keep along with each delinquency bit. However, that is not the whole story, since both the acquire and the reset-bit message need only reach a quorum of machines, and since those quorums may not intersect, there is no guarantee that an acquire from session S1 puts its unique id on to a delinquency bit, but the spawned reset-bit message never actually reaches that bit. Therefore, a subsequent acquire may also see the delinquent bit and attempt to tag it with its unique id, breaking the 1-acquire-per-session rule. We handle this case as follows, each bit keeps one slot per remote session, and thus the second acquire from S1 that will have to overwrite the unique id of the previous acquire of S1.

4.3 System implementation

4.3.1 How can our RDMA ZAB be 1000x faster than Zookeeper?

Our ZAB being 1000x faster than Zookeeper is not actually that surprising; Zookeeper uses TCP, is not multi-threaded and often goes to disk (for very non-technical reasons, as the authors describe in their ZAB paper [22]). We isolate the protocol in Zookeeper (i.e. ZAB) and make the most efficient implementation (in C with multi-threading, over MICA, with high-end RDMA NICS etc.).

4.4 General

4.4.1 What are Kite use-cases?

Kite offers a read-write-RMW API much like shared memory or Zookeeper. Use-cases: 1) Kite would be an excellent candidate to replace the consistency protocol of coordination services. E.g., strip ZAB from Zookeeper and plug-in Kite. 2) Kite can replace Key-Value Stores with multiple consistency levels that abound in industry (e.g. Yahoo Pnuts, Twitter Manhattan, Microsoft Pileus etc.) when deployed on an asynchronous environment. Such KVSes typically burden the programmer with ad-hoc mechanisms to choose the consistency for each access — in contrast RC provides a structured, well-defined approach to synchronization that the programmer already deals with during their everyday C/C++/Java development. 3) Kite can run fault-tolerant shared memory programs, such as any lock-free data structure, in a distributed highly-available manner.

4.4.2 Can Kite be sharded?

Not yet. But we note that Kite shards can be composed in a similar manner to Zookeeper [14]: by placing sync instructions when alternating among shards. The main difference is that in Kite, sync instructions need only be placed when synchronization is used.

References

[1] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. 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, pages 487–498, New York, NY, USA, 2011. ACM.

[2] Dotan Barak. Tips and tricks to optimize your RDMA code. https://www.rdmamojo.com/2013/06/08/tips-and-tricks-to-optimize-your-rdma-code/, June 2013. (Accessed on 19/04/2019).

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

[4] Jiaqing Du, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 4:1–4:13, New York, NY, USA, 2014. ACM.

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

[6] Heidi Howard. Distributed consensus revised. Technical Report UCAM-CL-TR-935, University of Cambridge, Computer Laboratory, April 2019.

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

[8] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, Boston, MA, 2019. USENIX Association.

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

[10] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 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, pages 185–201, Berkeley, CA, USA, 2016. USENIX Association.

[11] Christoph Lameter. Effective synchronization on Linux/NUMA systems. 2005.

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

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

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

[15] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 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, pages 429–444, Berkeley, CA, USA, 2014. USENIX Association.

[16] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 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, pages 401–416, New York, NY, USA, 2011. ACM.

[17] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger Semantics for Low-latency Geo-replicated Storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 313–328, Berkeley, CA, USA, 2013. USENIX Association.

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

[19] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. 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, pages 453–468, Berkeley, CA, USA, 2017. USENIX Association.

[20] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 358–372, New York, NY, USA, 2013. ACM.

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

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