USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

BODEGA: Localized Linearizable Reads at Anywhere Anytime via Roster Leases

Guanzhou Hu, *Amazon Web Services;* Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau, *University of Wisconsin–Madison*

https://www.usenix.org/conference/osdi26/presentation/hu-guanzhou

This paper is included in the Proceedings of the 20th USENIX Symposium on Operating Systems Design and Implementation.

July 13–15, 2026 • Seattle, WA, USA

ISBN 978-1-939133-55-7

Open access to the Proceedings of the 20th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

ARTIFACT EVALUATED
USENIX
AVAILABLE

Bodega: Localized Linearizable Reads at Anywhere Anytime via Roster Leases

Guanzhou Hu*
Amazon Web Services
Seattle, WA, USA
josehgz@amazon.com

Andrea C. Arpaci-Dusseau
University of Wisconsin–Madison
Madison, WI, USA
dusseau@cs.wisc.edu

Remzi H. Arpaci-Dusseau
University of Wisconsin–Madison
Madison, WI, USA
remzi@cs.wisc.edu

Abstract

We present Bodega, the first consensus protocol that serves linearizable reads locally from any desired node, regardless of interfering writes. Bodega attains this capability via a new notion of cluster metadata called the *roster*, which is a generalization of leadership; it tracks arbitrary subsets of replicas as *responder* nodes for local reads. A consistent agreement on the roster is established through *roster leases*, an all-to-all leasing mechanism that generalizes existing all-to-one leasing approaches (Leader Leases, Quorum Leases), unlocking a new point in the protocol design space. Bodega further employs *optimistic holding* and *early accept notifications* optimizations to minimize interruption from interfering writes, and *smart roster coverage* and *lightweight heartbeats* to maximize practicality. Bodega is a non-intrusive extension to classic consensus; it imposes no special requirements on writes other than a responder-covering quorum.

We implement Bodega and related works in Summerset, a protocol-generic replicated key-value store written in async Rust. We compare it to previous protocols (Leader Leases, EPaxos, PQR, and Quorum Leases) and two production coordination services (etcd and ZooKeeper). Bodega speeds up average client read requests by 5.6x~13.1x on real WAN clusters versus previous approaches under moderate write interference, and closely matches the performance of sequentially-consistent etcd and ZooKeeper deployments across YCSB workloads. Bodega supports fast proactive roster changes and delivers on-par write performance. Our Summerset codebase is open-sourced at https://github.com/josehu07/summerset/tree/bodega-artifact.

ACM Reference Format:
Guanzhou Hu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2026. Bodega: Localized Linearizable Reads at Anywhere Anytime via Roster Leases. In *Proceedings of the 20th USENIX Symposium on Operating Systems Design and Implementation (OSDI '26).* ACM, Seattle, WA, USA, 39 pages. https://doi.org/xxxxxxx.xxxxxxx

1 Introduction

Paxos-based consensus [60, 61, 65] (and protocols alike [85, 87]) serves as a critical foundation for modern distributed systems infrastructure. Originally used in limited contexts, e.g., for bespoke configuration information as in Petal [67], for critical metadata stores such as etcd [35, 97] and FireScroll [36, 95], and for lock services such as Chubby [21], consensus now forms the foundation of widely used cloud-native databases such as Spanner [30], CockroachDB [107], TiDB [50], ScyllaDB [100], and Physalia [19].

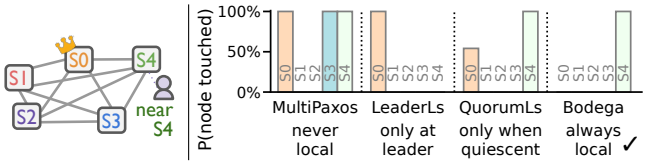


Figure 1. Frequency of touching a node on the critical path of reads *by a client near S4, in a cluster where S0 is the leader, with infrequent writes. Ideal outcome is 100% at S4.*

In these systems, simple access semantics are critical, enabling scalable services to be readily built atop them [4, 15]. In particular, *linearizability* is a strong consistency level they strive to provide: for interrelated requests, clients observe a real-time serial ordering, as if talking to a single node [47, 49].

Local Linearizable Reads. Delivering high performance in linearizable systems remains a daunting challenge. In the cloud era, systems replicate critical data across multiple geographically-distinct availability zones [7, 28, 79], to guard against correlated failures caused by power outage, fire, natural disaster, or operator error [22, 24, 43, 108]. By spreading replicas globally, robustness is achieved, but at the cost of performance due to quorum round trips [66].

The physical distribution of replicas yields an opportunity to serve read requests locally from a client's nearest replica. Reads comprise a majority of the workloads [8, 29, 88, 91]; achieving local reads can greatly reduce read latency and drastically increase overall throughput.

Existing Solutions Fall Short. Existing consensus protocols have demonstrated effective wide-area optimizations, but none, to our knowledge, supports coherently fast linearizable reads for workloads containing even small amounts of interfering writes. Leaderless protocols [27, 56, 62, 76, 80, 98] allow near quorums but not local reads. Others explore flexible quorums [42, 46, 48, 64, 106], utilize special hardware or client validation [5, 32, 40, 69, 93, 101, 112], or exploit API semantics favoring writes [37–39, 74, 86, 90, 96, 114]. Read leases (covered later) [10, 18, 25, 26, 41, 81, 82] are so far the most compelling, but are only effective at the leader or during quiescent periods without interfering writes.

*Work was done when at UW–Madison.

Self-Containment Necessitates Leases. A primary challenge of designing consensus protocols is self-containment: they cannot assume an external oracle. To enable local linearizable reads, the protocol must judge whether replica-local data is the most recent; contacting an external service to obtain this information would defeat the initial purpose. Moreover, having external dependencies would reduce the system's fault tolerance guarantees to those of the external services [57, 95, 107].

Within the design space of self-contained protocols, *leases* are a vital and powerful primitive. They carry timed promises that naturally tolerate faults through expiration [41], while only requiring bounded clock drifts (typical in today's cloud environments [40, 53, 68, 77]). This opens the gateway to local linearizable read protocols.

Leases Were Not Fully Exploited. Existing lease-infused protocols, however, do not employ the most suitable types of promises for local reads, and thus cannot fully realize their potential. As a motivating example, Figure 1 shows a 5-site cluster where S0 is the leader (and S4 is a local-read-enabled replica, if eligible), and reports the frequency of servers being touched by read requests from a client near S4. The workload contains 99% reads and only 1% writes, which favors existing approaches. Classic consensus (MultiPaxos) requires a majority accept quorum around S0. Leader Leases only protect stable leadership, so S0 can reply to reads directly, yet the delay between client and S0 persists. Quorum Leases allow granting leases to followers but use leases to guard against individual writes, rendering them vulnerable to even small amounts of concurrent writes to the key. As a result, a significant portion are redirected to the leader S0.

Our Approach: Roster Leases. We introduce the notion of a *roster*: a generalized form of cluster metadata that dictates not only leadership but also an assignment of local-read-enabled replicas (called *responders*) for arbitrary keys. Accordingly, we introduce *roster leases*, a novel all-to-all generalization of leader leases, deployed off the critical path to protect the agreement on the roster with no observable overhead. Roster leases stay valid in the absence of failures or proactive changes.

We present Bodega, a consensus protocol that uses roster leases to empower local linearizable reads. Bodega assures that writes never commit before reaching all of the key's active responders. A responder that holds a majority of leases can thus serve reads directly from its local state, if it knows the latest value will commit. When unsure, the responder *optimistically holds* the read locally until enough information is gathered, optionally utilizing *early accept notifications* to accelerate the hold. The roster may be changed manually by users, automatically according to runtime statistics, or in reaction to failures.

Our evaluation shows that Bodega speeds up client read requests by 5.6x~13.1x versus previous approaches under slight write interference, delivers comparable write performance, supports proactive roster changes in two message rounds as well as self-contained fault tolerance via leases, and matches the performance of sequentially-consistent etcd [35] and ZooKeeper [52] deployments across all YCSB variants.

Summary of Contributions. ❶ We introduce the notion of roster and propose Bodega, a consensus protocol equipped with a novel all-to-all roster leases algorithm, enabling local linearizable reads from any node at any time. ❷ We provide a thorough qualitative comparison across wide-area linearizable read approaches. ❸ We implement Bodega and related protocols in Summerset, a protocol-generic replicated key-value store, with 25.6k lines of async Rust. ❹ We evaluate Bodega comprehensively against previous works and two production coordination services, etcd and ZooKeeper, on 5-site wide-area clusters, delivering aforementioned evaluation results. ❺ We provide a formal TLA$^+$ specification of the full algorithm in the appendix.

The rest of the paper is organized as follows: §2 provides background and discusses existing solutions; §3 presents the Bodega design; §4 gives a formal comparison and proof; §5 covers the implementation of Bodega in Summerset; §6 presents our evaluation; §7-8 add discussions and related work; §9 concludes.

2 Background and Motivation

We provide background on consensus and linearizable reads, discuss existing solutions, and derive our goals for Bodega.

2.1 Consensus & Linearizable Reads

We consider the typical consensus problem of reaching agreement across message-passing server nodes, where nodes can be fail-slow/stop and the network is asynchronous [60].

Following well-established practice, nodes agree upon an ordered *log* of commands to behave as a replicated state machine (RSM) [61, 99]. For clarity, we use key-value `Put`/`Get` commands. In practice, `Get`s map to read-only requests that only query but do not update state (hereby referred to as *reads*), and `Put`s map to all other requests (*writes*). We restrict our discussion to non-transactional commands [17]; these are out of the scope of this paper.

Linearizable Reads. The consistency level dictates what results are allowed to be observed by clients [49]. *Linearizability* is the strongest non-transactional consistency level, where clients expect a serial ordering of commands with the real-time property: a command on key k issued at physical time t must be ordered after all the interfering writes on k acknowledged before t, and observe its latest value [6, 47, 80]. This semantic is mandatory for critical use cases such as metadata storage and coordination [19, 21, 35, 36], and is generally desirable as clients would otherwise get stale reads from the past (*sequential consistency*) [6, 59] or weaker guarantees [102, 109], complicating development. Linearizable reads are the primary focus of this paper.

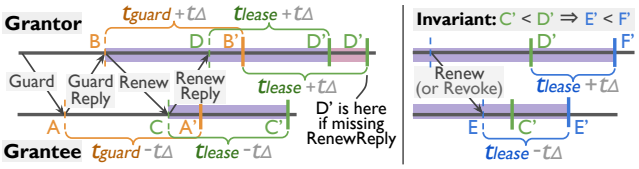


Figure 2. Demonstration of standard leasing. *Left: the guard phase helps establish the first iteration of promise coverage safely; grantee welcomes the first* `Renew` *only if it is received within the guarded period (C < A'). This allows the grantor to derive a safe expiration D' = B'+t_{lease}+t_Δ even if the* `RenewReply` *is lost, such that C' < D'. Right: grantor attempts to extend the promise with a* `Renew` *(or to actively revoke it with a* `Revoke`*), but has not received the grantee's reply. The leasing logic assures that E' < F'; therefore, if the grantee indeed failed, after F' the grantor can safely assert lease expiration. The purple-colored ranges depict when the lease is considered granted/held by the corresponding party.*

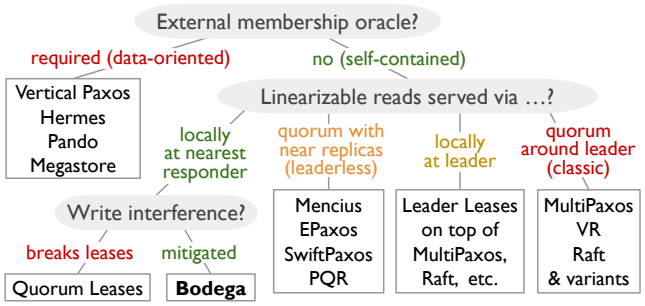


Figure 3. Categorization chart of relevant protocols. *Ideal properties for local reads are marked in green. See §2.3.*

Availability Requirements. A practical protocol must also offer *availability* for fault tolerance, allowing client requests to proceed under any minority number of node/network faults, and retaining consistency in all circumstances [49, 61].

2.2 Distributed Lease

Leases are a common distributed system technique [41]. They may be deployed as user-facing APIs through locks [21] and TTL-tagged objects [35], or as protocol-internal optimizations; we focus on the latter.

A lease is, conceptually, a directional limited-time *promise* that a *grantor* node makes to a *grantee*. It relies on bounded clock speed drift between the two ends, that is, over a given physical expiration time t_{lease} elapsed, the two nodes' clocks do not deviate more than a small t_Δ (no clock drift). This is typically true in today's distributed system environments [40, 53, 68, 77]; note that it does not assume synchronized clock timestamps [10, 30, 70] (no clock skew).

Standard One-to-One Leasing. The procedure of activating one lease between two nodes consists of an initial *guard* phase and repeated *renew* phases, depicted in Figure 2. The guard phase (left half) helps establish the first iteration in the presence of clock skew assuming bounded clock drift, and the renew phases (right half) keep it refreshed periodically [41, 81, 82]. Grantor starts to consider the lease granted when the first `Renew` is sent out (not when the `Guard` is sent), and grantee starts to consider the lease held when the first `Renew` is received (not when the `Guard` is received), using carefully designed timer durations as shown in Figure 2. The goal is to maintain this invariant: the grantor-side expiration time is *never earlier than* the grantee-side. A lease is considered held by the grantee when its clock has not surpassed $t_{\text{lease}}-t_\Delta$ after the last renewal received. The grantor can proactively deactivate the lease with a `Revoke` or, in the case of unresponsiveness, wait for $t_{\text{lease}}+t_\Delta$ without granting to let it safely expire.

2.3 Previous Work on Read Optimizations

Figure 3 presents a coarse-grained categorization of previous approaches to linearizable reads. The following sections discuss them in the general order from right to left.

2.3.1 Classic Protocols & Leader Leases

Protocols such as **MultiPaxos** [61], **VR** [85], and **Raft** [87] are the de-facto standards implemented in the wild [30, 35, 50, 103, 107]. While stale read options exist [25, 86], normal reads are treated obliviously just like other commands.

With MultiPaxos terminology, a typical protocol is as follows. A node S first makes a "covering-all" `Prepare` phase to settle a unique, higher *ballot* number for all non-committed *instances* (i.e., *slots*) in the log, effectively stepping up as a leader. Without competing leaders, S takes a client command, assigns the next vacant log slot, broadcasts `Accept` messages, and waits for $\geqslant m = \lceil \frac{n}{2} \rceil$ `AcceptReplys` with matching ballot including self (where n is an odd cluster size), after which the slot is marked committed and `Commit` notifications are broadcast asynchronously as announcement. S executes the commands in contiguously-committed slots in order and replies to their clients.

Leader Leases [25] are a commonly deployed optimization to establish *stable leadership*. All nodes grant lease to the most recent leader they are aware of (including self) after invalidating any old lease given out. If a leader S is holding $\geqslant m$ leases, it can safely assert that it is the only such leader in the cluster, i.e., the stable leader. Therefore, S (and only S) can reply to read requests locally using the latest committed value, knowing that no newer values could have committed.

2.3.2 Leaderless or Multi-Leader Approaches

Leaderless (or multi-leader) protocols distribute the responsibilities of a leader onto all nodes, improving scalability and latency under wide-area settings by allowing a fast-path quorum nearer to the clients. However, they are sensitive to command interference and often make local reads infeasible without degrading back to a leader-based protocol.

Mencius [76] assigns the leader role Round-Robin across nodes based on slot index. This mainly benefits scalability.

EPaxos [80, 104] absorbs the idea of inter-command dependencies from **Generalized Paxos** [62] and lets any node to act as the *command leader* for nearby clients. Nodes attach to each command its dependency set; without concurrent conflicting proposals, consensus can be reached on the fast path of `PreAccepts` by a (super-)majority quorum. Conflicts in dependencies require a second phase to resolve. Local reads are inherently hard to achieve in such a protocol without degrading to a leader-based protocol on keys of interest [104].

SwiftPaxos [98] improves EPaxos slow path to 1.5 RTTs (vs. 2) by re-introducing a leader in the slow phase.

PQR [27, 42] applies EPaxos-like leaderless optimization to only reads and not writes. Clients broadcast read requests directly to the nearest majority of servers. If all replies contain the same latest-seen value, all in committed status, then this value must be a valid linearizable read result. Otherwise, the client starts a *rinse* phase where it repeatedly polls arbitrary servers for commit confirmation of the value.

WPaxos [1] is a multi-leader design that partitions the key space and assigns a different leader per partition, improving locality of regional hot keys. However, it expects workload concentration and does not offer local reads at multiple replicas. Its adaptive partitioning approach can be applied orthogonally to consensus protocols including Bodega.

2.3.3 Enhanced Read Leases

Several works explored enhancements to *read leases* beyond stable leadership, enabling broader local reads.

Megastore [10] grants read leases to all replicas by a standalone coordinator. These leases carry the promise of not permitting any writes to covered keys. When writes arrive, leases are actively revoked (requiring an extra round-trip to all replicas) and local reads at followers are disabled until leases are re-granted. Megastore leases cover either all replicas or none; they also require external coordination and experience long downtimes during concurrent writes.

Quorum Leases [81, 82] extend read leases to configurable subsets of replicas. Leases are granted by replicas themselves, removing the need for an external coordinator. Upon writes, revocations are merged with the natural `Accept` messages and their replies, avoiding extra round-trips for writes in failure-free cases. Quorum Leases improve the configurability and write performance aspects of Megastore, but three insufficiencies with reads remain. ❶ Lease actions remain on the critical path, leading to frequent interruptions from writes. ❷ When fast-path local reads fail during lease downtimes, they are redirected to the leader or retried indefinitely by clients, leading to suboptimal slow-path latency. ❸ Assignment of grantees is configured with normal consensus commands, making failure cases hard to reason about and implement. Bodega eliminates all three insufficiencies via the powerful mechanism of background roster leases, as will become clear in §3.

2.3.4 With External Coordination

Protocols below assume external coordination.

Hermes [56] is a primary-backup replication protocol inspired by cache coherence protocols. It allows reads to be completed by individual nodes assuming that writes reach all nodes and are resolved synchronously with respect to each other (similar to CPU shared cache invalidation). Hermes inherits its architectural assumption from **Vertical Paxos** [64], requiring an external membership manager for reconfigurations upon failures.

Pando [106] is a WAN-aware, erasure-coded protocol that emphasizes cost efficiency. It allows statically tunable read-write quorums, which are settable ahead-of-time before deployment, and assumes a network topology with frontends and an external service for membership management.

2.4 Summary of Goals

After reviewing existing solutions, we summarize the desired properties of a linearizable read protocol as our design goals:

- *Self-contained*: no external metadata oracle dependencies.
- *Local reads anywhere*: enable local linearizable reads at arbitrary subsets of replicas as appropriate.
- *Local reads at anytime*: keep reads localized during concurrent interfering writes, minimizing degradation time and maintaining good slow-case latency.
- *Configurable*: tunable against arbitrary ranges of keys.
- *Non-intrusive*: designed atop classic consensus, introducing marginal performance impacts on writes and retaining availability under any minority number of failures.

Via these goals, Bodega delivers superior performance characteristics compared to aforementioned approaches. We will show them both theoretically (§4) and experimentally (§6).

3 Design

In this section, we present the core design of Bodega.

Design Outline. We derive the complete design in three steps: ❶ define the roster, ❷ design optimal normal case operations assuming replicas agree on the same stable roster, and ❸ introduce all-to-all roster leases, the enabler behind the fault-resilient agreement on the roster.

For clarity, we adhere to Paxos-style terminology throughout this paper. All the optimizations are applicable to Raft-style protocols due to their fundamental duality [110].

3.1 The Roster

We start by introducing the core concepts behind Bodega: responder status and the roster. A node is a *responder* for a key if it is expected to serve read requests on that key locally without actively contacting other nodes. A *roster* is the collection of each node's desired capabilities at a certain time; specifically, it dictates:

- The node ID of the current leader node.
- For each (range of) key(s): the node IDs of its responders.

The roster is a generalization of leadership from classic protocols: besides the one special leader role, we now have special responder roles for selected keys. The leader can be implicitly treated as a responder for all keys, and different keys can additionally mark different nodes as responders.

The optimal choice of responders for each key depends on various factors: ❶ client locations and proximity, ❷ workload read-heaviness and skewness, and ❸ cluster topology and status. This paper focuses on the mechanisms supporting the roster rather than the policies for tuning it; we recognize that the latter could be an intriguing study on its own.

The system starts from an empty roster with a null leader ID and an empty responder set for the entire keyspace. Every newly-proposed roster is associated with (and identified by) a unique ballot number, forming a $\langle bal, ros \rangle$ pair, where bal is the ballot number formed by concatenating a monotonically increasing integer b with the proposing node's ID r to ensure uniqueness. Rosters of different ballots may contain the same content but are still considered different. Roster changes may happen due to explicit requests by users, automatic tuning from statistics, or mandatorily in reaction to failures.

3.2 Normal Case Operations

We first describe normal case operations, using Figure 4 as a demonstrative example. In a 5-node cluster, S0 is the leader (depicted by the crown) and S2,3,4 are additional responders (depicted by the red star symbols) for a specific key k. Assume, in this section, that this is the latest roster all nodes know and consider stable according to leases. Nodes use their known roster to assure that writes to k would never commit before reaching all of its active responders. A responder can therefore serve reads directly if it knows the latest value of k will commit; when unsure, optimizations exist.

3.2.1 Writes

Writes follow the same leader-based process as in MultiPaxos (Figure 4 blue arrows), except for an updated commit condition. Normally, a write to key k can be marked as committed and acknowledged once $\geqslant m$ `AcceptReplys` are received. We impose an additional constraint that it must also have received replies from all the responders for k, according to the leader's current roster.

Requiring a write quorum that covers all responders is an unavoidable penalty that any local linearizable read algorithm must pay. Luckily, without far-off responders, this penalty is marginal as the system usually picks a leader with relatively uniform distances to other replicas, and the write must anyway reach a majority. This aligns with previous observations [81] and our evaluation results (§6). Distant responders could still be appropriate for certain workloads.

3.2.2 Reads

Clients send read requests on key k to the closest responder server for k (Figure 4 green arrows). It is common for clients of wide-area systems to be co-located with some replica; for example, consensus is usually part of an outer system, e.g., a database, where requests come directly from participating sites, but this is not a requirement.

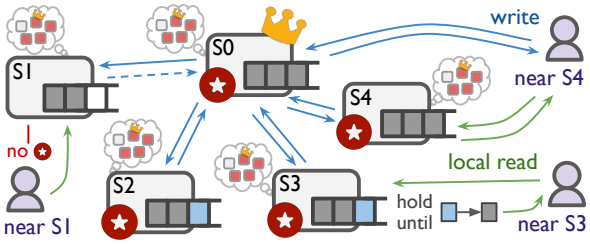


Figure 4. Normal case operations of Bodega. *Assume all nodes agree on the same example roster: S0 is the leader and S0,2-4 are responders for a key while S1 is not. See §3.2.*

When a server S (with a stable roster) takes a read, there are three cases. ❶ S is the leader, in which case S simply finds the highest committed slot in its log that contains a write to k and returns the value. The leader does not need to worry about in-progress writes [25]. ❷ S is neither a leader nor a responder for k (e.g., S1 in Fig. 4), in which case S rejects the read and promptly redirects the client to another server, preferably a close-by responder or the leader. ❸ S is a non-leader responder. In this case, S looks up the highest slot in its log that contains any write to k. If the slot is in committed status, S immediately replies with the value (e.g., the read at S4 in Fig. 4); otherwise, S cannot yet determine whether that value will surely commit or will be overwritten due to impending failures (e.g., the read at S3), in which case S optimistically *holds* the read.

Optimistic Holding. S expects its connection with the leader to be normally healthy, hence anticipates quick commitment for in-flight writes. It is usually faster and more efficient for S to withhold local reads that depend on an in-flight write and to reply as soon as commitment is known, than rejecting the reads and letting clients redirect or retry. In failure-free cases, the commit notification for an interfering write arrives in at most one RTT (from when S receives the `Accept` from the leader and replies to it). Note that even with a constant stream of writes, held reads are not blocked indefinitely: they are released as soon as their associated slot turns committed.

A responder S optimistically holds a local read by adding it to a *pending set* attached to the corresponding slot. Upon receiving the commit notification for a slot, S releases the pending reads and replies with the committed value. To handle cases where the leader fails to notify S promptly, clients start an *unhold* timeout when sending local read requests; if the timeout is reached, clients proactively issue the same request to another responder or the leader (with the same req ID, which is safe since reads are idempotent) and use the earliest reply. A good timeout length is longer than the usual RTT between S and the current leader.

Early Accept Notifications. BODEGA can optionally incorporate a further optimization that reduces average holding time when under low load: followers not only reply to `Accepts` on key k to the leader, but also broadcast notifications to k's responders. Once a responder S has received m notifications (counting self), it can assert that a pending slot will surely commit even across minority failures. A similar optimization exists for BFT writes [23]. On average, this halves the expected holding time for interfered local reads. This optimization is optional and can be turned off when write traffic increases.

3.3 Roster Leases

So far, we have assumed a consistent roster across the cluster without showing how it is achieved. The idea is to exchange roster leases in an all-to-all manner, between at least a majority of nodes and all nodes that may be responders for some key. When holding a majority number of leases, responders know that the roster is stable, and the leader (as an implicit responder) also knows the identity of all responders and will not commit writes without notifying them.

	Lease Primitive	Leader Leases	Roster Leases
Pattern	one-to-one	all-to-one	all-to-all
Safe #grantors	1	m	m
Must to whom	-	leader	responders

We present how BODEGA deploys off-the-critical-path *roster leases* to establish a stable roster elegantly and efficiently. We use Figure 5 as an illustration when needed.

Lease-related States. Besides the SMR log and the $\langle bal, ros\rangle$ pair, we let every node S act as both a lease grantor and a grantee (recall §2.2 for how a standard lease grant primitive works). This means S maintains the following additional data structures: ❶ two lists of grantor-side timers $T_{\text{guardTo},\,p}$ and $T_{\text{renewTo},\,p}$ per peer node P; ❷ two corresponding sets $\{\}_{\text{guardTo}}$ and $\{\}_{\text{renewTo}}$ for tracking which peers are S currently guarding/renewing to; ❸ two lists of grantee-side timers $T_{\text{guardBy},\,p}$ and $T_{\text{renewBy},\,p}$ per peer P; ❹ two sets $\{\}_{\text{guardBy}}$ and $\{\}_{\text{renewBy}}$ for tracking the guards/renewals S currently holds; ❺ a list of safety slot numbers $thresh_p$, that specifies the highest slot S has accepted from each peer P.

3.3.1 Roster Leases Activation

We first describe how roster leases are activated. Consider node X wants to announce a new roster ros'; this could be due to, e.g., stepping up as new leader (by setting X as the leader in ros') or other reasons covered in §3.3.2. X composes a unique, higher ballot bal' by concatenating $(b+1)$ with its node ID, where b is the higher part of the current bal. X then broadcasts the $\langle bal', ros'\rangle$ pair to all nodes including self.

For any node S upon receiving a ballot bal' higher than ever seen, it first ensures all old leases are safely revoked or expired (discussed later in §3.3.2). Then, it moves on to $\langle bal', ros'\rangle$ and starts a `initiate_leases(`bal'`)` procedure, where it begins granting leases for the new roster to all peers asynchronously in parallel.

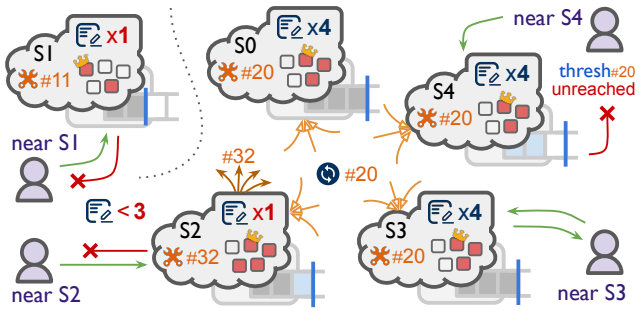


Figure 5. All-to-all roster leases demonstrated. *S0,3,4 are each holding ≥ majority grants of roster #20; S4 has not seen all slots up to #20's safety threshold. S1 is stuck with an older roster of #11. S2 is initiating a new roster of #32. See §3.3.*

To each peer P, the procedure obeys standard lease granting: S and P first complete the guard phase, exchanging a sequence of `Guard`, `GuardReply`, `Renew`, and `RenewReply`, and utilizing proper timers along the way. If all goes well, S should have P in its $\{\}_{\text{renewTo}}$ and have T_{renewTo} properly extended; it repeats renewals periodically to keep the S-to-P lease refreshed. Similarly, P should have S in its $\{\}_{\text{renewBy}}$ and have T_{renewBy} kicked off properly. Whenever a $T_{\text{intent},\,p}$ times out for any intent among the four, the peer is removed from the corresponding set $\{\}_{\text{intent}}$, leading to a retry of the guard phase or a proposal of a new roster.

After transitioning to ros', if S sees itself being the leader of ros', it does the usual step-up routine of redoing the `Prepare` phase for non-committed slots of its log.

Stable Condition & Safety Thresholds. As shown in §2.2 and above, a node P is considered granted a lease by S when S ∈ P's $\{\}_{\text{renewBy}}$ set. Assume P itself is always in the set. The size of this set, $|\{\}_{\text{renewBy}}|$, indicates the number of lease grants P currently holds. When $|\{\}_{\text{renewBy}}| \geqslant m$, then P knows at least a majority number of nodes in the cluster has the same latest $\langle bal, ros\rangle$ as P and that at most one such roster exists; this is called the *stable* roster of the cluster and is a necessary precondition for all optimizations described in §3.2. For example, in Figure 5, the local reads at S1 and S2 are rejected due to an insufficient lease count.

This condition alone is not enough, though. When a node directly inspects its log and uses the highest slot index for local reads, it is assuming that its log is up to date and contains all the recently committed instances; this is usually true, but could be violated during the short period of time when a node that has fallen behind joins a new roster. To address this, a node should be informed of other peers' acceptance progress when transitioning to a new roster.

We let `Guard` messages from S to P carry an extra number, which is the highest slot number that S has ever accepted. P stores the number in its *thresh* list. A node permits local reads only if it has committed all the slots up to the m-th smallest slot number in its *thresh* list; otherwise, it might not have observed the latest committed writes yet. S4 in Figure 5,

for example, has not reached this condition and thus cannot start serving reads locally yet.

In summary, all the stable leader and local read operations of §3.2 are preceded by the following stable condition check:

$$|\{\}_{\text{renewBy}}| \geqslant m \quad (1)$$
$$\wedge \ \exists \text{ size-}m \text{ subset } E \subseteq \{\}_{\text{renewBy}} :$$
$$\text{committed all slots up to } \mathit{thresh}_p, \forall p \in E$$

If the check fails, the operation falls back to classic consensus as if it is a write, which does not require this check.

3.3.2 Revocation & Expiration

Most roster lease activations happen when there are ongoing old leases in the system. Broadly speaking, a roster change may be triggered by one of the following reasons.

- Node initiates a new roster in reaction to suspected failures, removing failed nodes from special responder roles.
- Node autonomously proposes a new, more optimized roster according to collected workload statistics.
- Node receives an explicit roster change request from user.

In either case, before `initiate_leases()`, a node S always invokes the `revoke_leases(`*bal*`)` procedure synchronously to ensure that all the leases it is granting or holding with the older ballot *bal* are safely revoked and removed. To do so, S clears its $\{\}_{\text{guardTo}}$ set and broadcasts `Revoke` messages carrying the old ballot. Whenever a node P receives a `Revoke` with matching ballot from S, it removes S from the $\{\}_{\text{guardBy}}$ and $\{\}_{\text{renewBy}}$ sets and replies with `RevokeReply`.

S either receives a `RevokeReply` from P promptly (common fast case) or has to wait for expiration timeout (failure case), after which it removes P from $\{\}_{\text{renewTo}}$. Note that, unless failures occur and force a wait on expiration, a roster change completes swiftly within two message rounds: one for the revocation and the other for the initiation guards.

3.3.3 Piggybacking on Heartbeats

Distributed systems already deploy heartbeats for tasks such as health tracking [35, 52, 80, 94]. This opens the opportunity to enable roster leases without common-case overheads, by piggybacking lease messages onto existing periodic heartbeats. Bodega piggybacks all the `Renew` and `RenewReply` messages onto heartbeats, and uses a proper heartbeat interval $t_{\text{hb\_send}}$ such that leases are refreshed in time. Heartbeat messages also carry the sender's $\langle \mathit{bal}, \mathit{ros} \rangle$ pair to let receivers discover roster changes.

Each node has per-peer timers $T_{\text{heartbeat},\,p}$ which are used for promptly detecting failures from peers; a peer is considered down if no heartbeats were received from it for $t_{\text{hb\_fail}}$. A rule of thumb for choosing good timeout lengths for a cluster is:

$$\text{avg. RTT} < t_{\text{hb\_send}} \ll t_{\text{hb\_fail}} < t_{\text{guard}} = t_{\text{lease}} \quad (2)$$

Bodega uses the following defaults for wide-area replication: $t_{\text{hb\_send}} = 120\text{ms}$, $t_{\text{hb\_fail}} \approx 1200\text{ms}$, $t_{\text{guard}} = t_{\text{lease}} = 2500\text{ms}$.

Protocol	W	R	R^*	D^*	⚡	⬚
Leader Ls	$l+m$	l	l	-	●	○
EPaxos	$c+M$	$c+M$	$c+M+m$	M	●	○
Hermes	$c+N$	c	$c \sim c+\frac{N}{2}$	$\frac{N}{2}$	◐	○
PQR	$l+m$	$c+m$	$c+m*\text{rinses}$	m	●	○
PQR (+ Ldr Ls)	$l+m$	$c+m$	$c+m+l$	m	●	○
Pando	$c+m$	$c+m$	$c+N$	N	◐	◐
Megastore	$l+2N$	c	$c+l$	$2N$	◐	○
Quorum Ls	$l+N$	c	$c+l$	$2N$	●	●
Qrm Ls (passive)	$l+N$	c	$c+l$	N	◐	◐
Bodega	$l+N$	c	$c \sim c+\frac{m}{2}$	$\frac{m}{2}$	●	●

Table 1. Qualitative comparison across protocols *assuming the most read-optimized config of each protocol.* ***W****: write latency;* ***R****: read latency if quiescent;* ***R**** *: read latency if there is an interfering write;* ***D**** *: read performance degradation period length.* ⚡*: fault tolerance (without external oracle).* ⬚*: allows tunable rosters.* $\underline{l}$*: client-leader RTT,* $\underline{c}$*: client-nearest server RTT,* $\underline{m}$*: time to establish simple majority,* $\underline{M}$*: time to establish super majority,* $\underline{N}$*: time to form all-nodes quorum. Cell is shaded darker if value is higher.*

4 Formal Presentation and Proof

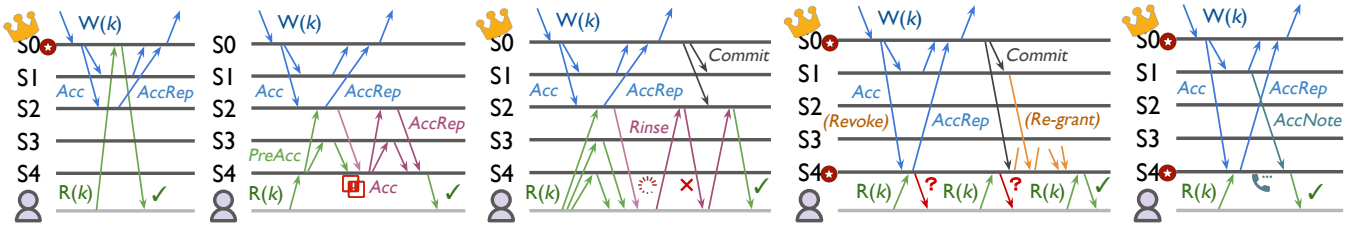
We provide a qualitative comparison across notable protocols (Figure 6, Table 1), a formal presentation of the Bodega algorithm (Figure 7), and a concise proof of correctness.

4.1 Comparison Across Protocols

In Table 1, we model the normal-case write and read latency, degraded read latency under write interference, and degradation period length of related protocols. Cells are shaded according to example values from the Figure 8(b) GEO setting (lighter is better). We also indicate whether a protocol retains the fault tolerance of classic protocols and whether it allows tunable configs. If tunable, we use its most read-optimized config that tolerates $f = \lfloor \frac{n}{2} \rfloor$ faults. Assume only one interfering write.

The following symbols are used to model performance. $\underline{l}$: RTT between client and the leader, $\underline{c}$: RTT between client and its nearest server, $\underline{m}$: time to establish a simple majority quorum (i.e., to reach majority nodes from some server and receive replies), $\underline{M}$: time to establish a super majority quorum (as in EPaxos [80]), $\underline{N}$: time to form a quorum composed of all nodes. For an average client in typical WAN-scale settings, one should expect $c \ll l \approx m < M < N$.

Most results are derived naturally from Figure 6, §2.3, and §3.1-3.3. We provide supplementary explanations. **PQR (+ Ldr Ls)** is a straightforward variant of PQR combined with Leader Leases; if a near quorum read attempt fails, the client contacts the stable leader directly, bounding slow-path latency by $c + m + l$. We assume Quorum Leases always incorporate Leader Leases. **Qrm Ls (passive)** is a variant of Quorum Leases where we deliberately let grantees keep



(a) Leader Leases **(b)** EPaxos **(c)** PQR **(d)** Quorum Leases **(e)** Bodega

Figure 6. Timeline comparison across protocols of linearizable reads *in the presence of an interfering write.* **W(k)***: write key k,* **R(k)***: read key k,* ***Acc****: Accept,* ***AccRep****: AcceptReply,* ***PreAcc****: EPaxos PreAccept, ⧉: EPaxos dependencies conflict,* ***Rinse****: PQR repeated poll.* ***Commit****: commit notification,* ***AccNote****: accept notification (optional), 📞: Bodega optimistic holding.*

the leases upon accept to show the upper bound of Quorum Leases performance, saving one re-granting RTT from the degradation time. Doing so risks blocking fault-induced roster change commands as described in §3.3. Hermes uses primary-backup broadcast and thus requires external coordination for fault tolerance; Megastore is similar. Pando uses a pre-deployment planner to dictate erasure coding and quorum composition. Bodega achieves the best across all metrics and retains fault tolerance and configurability.

4.2 Formal Presentation

Figure 7 presents a complete, code-oblivious summary of the Bodega algorithm, which can be used as a reference for implementors. We also provide a formal, model-checked TLA$^+$ specification in Appendix A.

4.3 Proof

We provide a proof of Bodega's local read linearizability and write liveness, assuming well-established results of the safety and liveness of leases [41]. For linearizability, only locally-served reads need proof, as Bodega behaves the same as classic consensus otherwise.

Linearizability. A local read R served by server S observes any write W acknowledged cluster-wise before R was issued.

Proof. R is served locally only if S is a responder that passes the stable roster check (1). Let the stable roster be $\langle bal, ros \rangle$.

Case #1: W was committed on a ballot $> bal$. It is impossible because the latest ballot on at least a majority is bal.

Case #2: W was committed on a ballot $= bal$. By the injective ballot-roster mapping and the commit condition of writes, S must be in W's write quorum and have W in its log.

Case #3: W was committed on a ballot $< bal$. By majority intersection, for any size-m subset $E \in$ S's $\{\}_{\text{renewBy}}$, at least one of the lease grantors $P \in E$ accepted W at its committed slot x before granting to S. This implies $thresh_p \geqslant x$. □

Liveness of Writes. A write W can always eventually make progress if retried on a majority group G of healthy servers.

Proof. By the property of leases, after old leases expire, a roster change can eventually be made on all servers $\in G$ to restrict the leader and all the responders to be contained in G. Then, normal consensus applies. □

5 Implementation

The Summerset KV-store. We develop Summerset, a distributed, replicated, protocol-generic key-value store. Summerset is written in async Rust/`tokio` using a lock-less architecture and serves as a fair codebase for evaluating consensus and replication protocols.

The codebase has 13.6k lines of infrastructure code and covers five protocols of interest: MultiPaxos w/ Leader Leases (2.5k), EPaxos (3.1k), PQR & variant (2.8k), Quorum Leases & variant (3.2k), and Bodega (3.0k). All protocol implementations have passed extensive tests. The full Summerset codebase is open-sourced at https://github.com/josehu07/summerset/tree/bodega-artifact.

5.1 Smart Roster Coverage

In cases where users desire local reads but cannot observe workload patterns externally, Bodega servers can collect statistics and automatically propose roster changes to mark servers as responders for proper keys. Our default implementation traces per-key read/write request counts grouped by clients' preferred nearby server IDs. For a key, if > 95% requests are reads at a periodic check, then servers near > 20% of the reads are added as responders. More sophisticated strategies exist; for example, straggler detection can help remove fail-slow nodes from responders promptly [51, 54].

5.2 Lightweight Heartbeats

In §3.3, we described roster leases as if all heartbeats carry the complete roster data structure. In practice, rosters with fine-grained key ranges can get large (tens of KBs). Luckily, most heartbeats in Bodega are *lightweight heartbeats*: the sender puts in only the ballot number to indicate that the roster has not changed from previous heartbeats. Full-sized heartbeats are sent when changes occur.

Similarly, clients may request a server to send the roster along with a command reply, and then cache this roster as a heuristic for choosing the best responder for local reads.

5.3 Other Practical Details

Request Batching. As is common practice, Bodega deploys request batching at servers (at 1 ms intervals) for non-local-read commands. Each log slot contains a batch of requests

Figure 7. Complete Summary of the Bodega algorithm. *Lists all actions a node r would take upon certain conditions, grouped by purposes for clarity: triggers for a new roster, granting procedure of new roster leases, heartbeats and lease renewals, handling client write requests, handling client read requests. The description is based on a regular key-value API. Nodes implicitly retransmit non-acked messages. Broadcast msg receivers include the sender itself. Clock drift between nodes is assumed to be bounded by t_Δ, as is required by any distributed lease algorithm; clock skews are irrelevant thanks to Guards. The arrows annotate a natural reading order that follows the usual flow of the protocol.*

and the commit condition is checked for all writes contained.

Snapshots. Bodega servers take periodic snapshots of the executed prefix of the log [87]. Local reads past the beginning of truncated log look up the latest snapshot directly.

Membership Management. Membership changes are handled identically to *reconfigurations* in other protocols [25, 80, 85], just with an extra step of proposing and stabilizing an empty roster with no responders ahead of the change.

6 Evaluation

We do comprehensive evaluations to answer these questions:

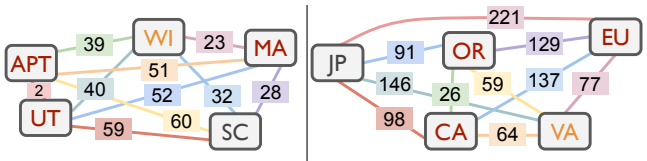
- How does Bodega perform compared to other protocols under microbenchmarks of various write intensities? (§6.1)
- What do the request latency distributions look like? (§6.2)
- What are the impacts of write interference, and how does performance change with write ratios & value sizes? (§6.2)
- How do roster changes impact performance? (§6.3)
- How does Bodega behave with different choices of responders and different coverages of keys? (§6.3)
- How does Bodega compare with production coordination services, etcd & ZooKeeper, under YCSB workloads? (§6.4)

Experimental Setup. All experiments are run on two CloudLab [33] clusters, hereafter called WAN and GEO, shown in Figure 8. WAN is a wide-area cluster spanning five CloudLab sites with nodes of similar hardware types: WI-`c220g5`, UT-`xl170`, SC-`c6320`, MA-`rs620`, and APT-`r320`. §6.1 also includes a GEO cluster of five `c220g5` nodes emulated with Google Cloud RTTs reported in previous work [104] using Linux kernel `netem` [71]. All nodes' public NICs have 1Gbps bandwidth. The orange-colored site in Figure 8 denotes the leader and the red-colored sites denote responders.

Clients are launched on machines evenly distributed across all datacenters, each marking the nearby server as their preferred server for local reads when eligible. All machines run Linux kernel v6.1.64 and pin processes to disjoint cores. All protocols use 120 ms heartbeat interval, 1200±300 ms randomized heartbeat timeout, and 2500±100 ms lease expiration (if applicable). All protocols send immediate `Commit` notifications: whenever a commit decision is made, `Commits` are broadcast to other servers promptly.

6.1 Normal Case Performance

We run microbenchmarks on both cluster settings and compare: ordinary MultiPaxos, Leader Leases, EPaxos, PQR, PQR (+ Leader Leases) variant, Quorum Leases, Quorum Leases (passive) variant, and Bodega. We spawn 50 closed-loop clients with 10 near each server and let all clients run a microbenchmark with 1k 8B-size keys and 128B values; keys are chosen uniformly. We test three write percentages in the workload mix: 0%, 1%, and 10%. Figure 9 shows the normalized throughput (w.r.t. Leader Leases), avg. read latency, and avg. write latency perceived by clients at different locations. Leader and responders are set as depicted in Figure 8. The



(a) WAN (CloudLab sites) **(b)** GEO (GCP [104], emulated)

Figure 8. Evaluation settings. *Orange denotes designated leader node and Red denotes other responders, if relevant. The edges mark per-pair RTT values in milliseconds. See §6.*

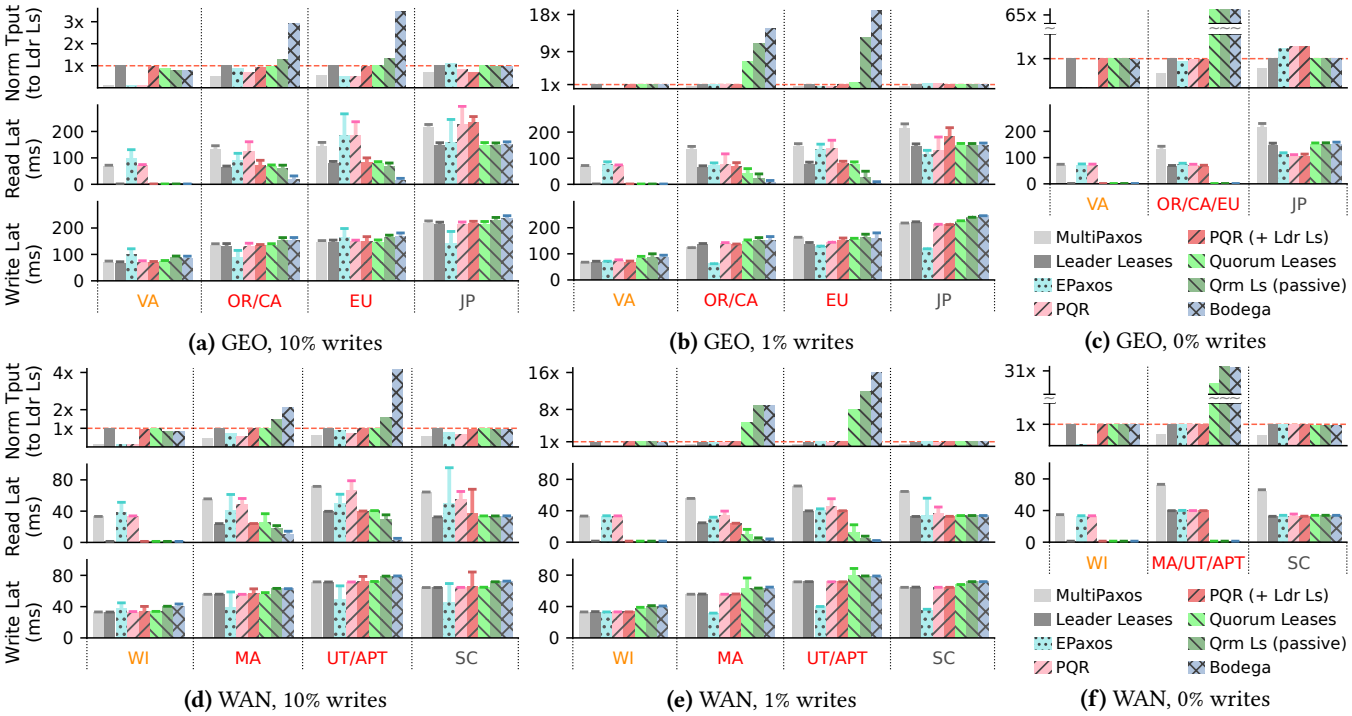
red dashed lines indicate baseline Leader Leases throughput, and the top Ts on latency bars indicate P99 latency.

The results yield the following observations. First, except for a few datapoints (which we soon discuss), both GEO and WAN clusters exhibit similar performance patterns, just with different absolute values due to RTT differences.

Second, for writes, all protocols except EPaxos exhibit similar performance. Quorum Leases and Bodega have higher write latency due to the requirement of writes reaching responders; this explains the small throughput gap between them and Leader Leases for near-leader clients with 10% writes. EPaxos delivers better average (but not P99) write latency due to its leaderless write protocol design.

Third, we observe coherent patterns for read performance. ❶ Compared to ordinary MultiPaxos, Leader Leases cut read latency for near-leader clients to nearly zero, but other clients still pay an RTT to the leader for reads. ❷ PQR (and its Leader Leases variant) show worse (or identical) performance compared to Leader Leases. The only exception is in the GEO, 0% writes setting for the JP clients; they are so far away from the leader that a nearer majority quorum actually helps, letting them outperform local read protocols (since JP is not marked as a responder). ❸ EPaxos has similar read performance as PQR but with higher P99 latency when there are writes. ❹ Both Quorum Leases variants and Bodega show the same performance as Leader Leases for clients near the leader or a non-responder. ❺ Quorum Leases and Bodega both deliver extraordinary read performance for clients near responders when with 0% writes. ❻ Bodega sustains this read performance advantage and keeps read latency close to zero for higher write intensities. In contrast, Quorum Leases performance quickly drops and almost degrades back to Leader Leases for 10% writes. This shows Bodega's resilience to write interference, a crucial advantage.

Throughput to Read Latency Curves. We take the 10% writes WAN setting of Figure 9(d) and run open-loop clients with varying request concurrency up to 100 reqs/sec to profile the throughput-latency curves, which we plot in Figure 10. The x-axis is aggregated throughput and the y-axis is average latency of all requests. The never-local-read protocols (MultiPaxos, PQR, EPaxos) form a cluster at the upper left as expected, due to unavoidable network traffic per read. Leader Leases reveal a throughput upper bound at ~1.8k where the leader node starts to overload; this bound



(a) GEO, 10% writes **(b)** GEO, 1% writes **(c)** GEO, 0% writes

(d) WAN, 10% writes **(e)** WAN, 1% writes **(f)** WAN, 0% writes

Figure 9. Normalized throughput and latency at different client locations *across different write intensities. See §6.1.*

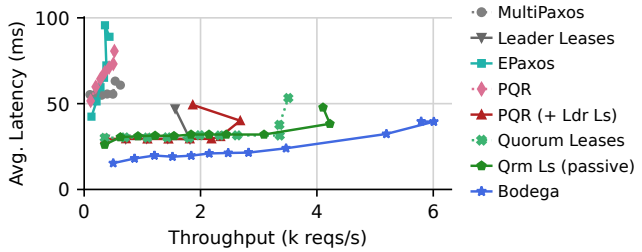


Figure 10. Tput to avg. latency curves *on WAN. See §6.1.*

is slightly higher for PQR + Leader Leases at ~2.2k. Quorum Leases see similar latency under low load and a higher throughput bound at ~3.4k, matching Figure 9. Bodega sustains 1.5x better latency because of most reads being served locally without fallbacks; throughput limit is accordingly higher at ~6k.

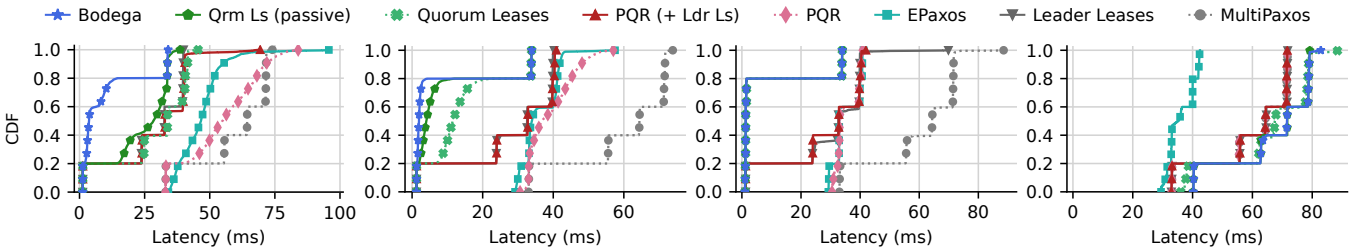
6.2 Detailed Performance Anatomy

We conduct a zoomed-in study across various workloads and metric dimensions.

Latency CDFs. We collect request latency CDFs across all 50 clients of the WAN setting (Fig.9(d)-9(f)) and plot them in Figure 11. Results are filtered to show a single key for a clean pattern. Each site contributes an equal 20% of datapoints.

We make four observations. ❶ Write latencies across all workloads are similar and are presented as one Figure 11(d). Quorum Leases and Bodega show slightly higher write latencies in favor of responder local reads, while EPaxos delivers the same level of latencies as its reads due to its leaderless design. These results align with §6.1. ❷ At 0% writes, all protocols deliver a read performance close to their theoretical best, though a few outlier datapoints remain. ❸ At 1% writes, slight write interference occurs. Quorum Leases reads deviate from Bodega, with the passive variant delivering roughly half the latency of the original variant. ❹ At 10% writes, differences in read latency distributions are the most obvious. MultiPaxos clients' read latency clearly correlates with their distance to the leader; Leader Leases are similar but with a majority-quorum latency subtracted. PQR and EPaxos exhibit suboptimal latency and have high tail latency of up to 100 ms for a read; this is due to the need for conflict resolution. Quorum Leases variants both degrade to Leader Leases. Bodega delivers outstanding local read performance as expected (except for the 20% non-local SC clients).

Visualizing Write Interference. We use a similar setting to the read-only workload in §6.1 on the WAN cluster, but this time with open-loop clients, each sending reads at a rate of 400 reqs/sec to a key. At ~15 secs, we let one client issue a write command to the key. We monitor the average read latency across the three non-leader responders for the local read protocol variants, and plot them over a time axis in Figure 12. We see that the write introduces an interruption to local reads for all three protocols. Both Quorum Leases variants degrade to 40 ms read latency, which is the average RTT to the stable leader; the passive variant shows a shorter degradation duration. Bodega ❶ shortens the degradation time to ~25 ms; recall Table 1, this corresponds to ~$\frac{m}{2}$, and ❷ allows all reads to be held locally and be released at the end of the degradation, leading to better latencies also for disrupted reads.



(a) Reads w/ 10% writes **(b)** Reads w/ 1% writes **(c)** Reads w/ 0% writes **(d)** Writes

Figure 11. Latency CDFs of requests *in the WAN setting across different write intensities, focusing on one specific key. See §6.2.*

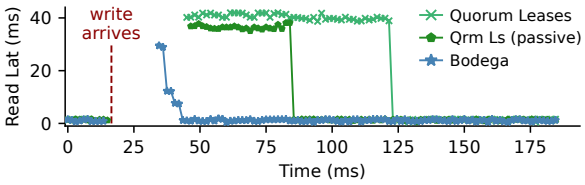


Figure 12. Read latency after an interfering write. *The x-axis is time at which the reads finish. See §6.2.*

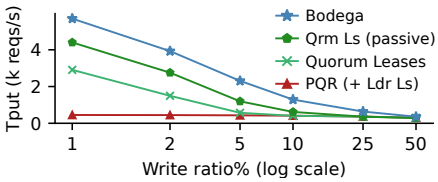


Figure 13. Throughput vs. write ratio. *The x-axis is log-scale (same for Fig. 14).*

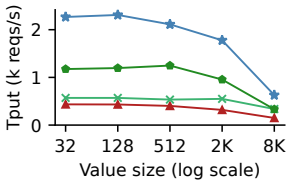


Figure 14. Throughput vs. value size. *See §6.2.*

Varying Write Ratios. We take the same setup as §6.1 on the WAN cluster and vary the write ratios of the workload mix from 1% to 50% while fixing value size to 128B. We report the aggregate throughput in Figure 13. All protocols except the PQR (+ Ldr Ls) baseline show a trend of lower throughput with higher write interference as local reads become less profitable. The results match Figure 9(d)-9(e).

Varying Value Sizes. We repeat the same setup as above and vary the value size while fixing the write ratio at 5%. As expected, Figure 14 shows that smaller values have little impact on performance, but throughput drops with larger values due to slower writes and larger read results to transfer.

6.3 Roster Changes & Composition

We evaluate roster change performance and roster coverage.

Impact of Frequent Failures (Simulation). We first justify roster leases' stability under frequent failures through an exaggerated simulation. We run a 10ms-timeslice-based Monte Carlo simulation with 10% writes using performance numbers measured in Figure 9(d). We apply a failure probability of 0.5% every second–much larger than real-world servers [3]–and a recovery probability of 1% every second. After any failure, the cluster delivers 3 seconds of zero throughput to simulate lease expiration; after any recovery of a responder, the cluster delivers 100ms of zero throughput to simulate a proactive roster change. Figure 15 shows aggregated throughput of 1 million simulated seconds. We can see that, despite these exaggerated conditions, Bodega still delivers up to 2.2x throughput compared to traditional Leader Leases.

Roster Change Duration. On the WAN cluster, we zoom in and compare the duration of two types of roster changes: failure-induced (where waiting for lease expiration is necessary) vs. regular (where revocations complete quickly). Regular roster changes finish in just two message rounds, because they are no more than a lease revocation followed by an initiation. We create an open-loop client near WI and let it issue a 50%-read workload at a rate of 400 reqs/sec to 1k keys. We plot real-time latencies in Figure 16.

At ~800 ms, we crash the UT node, which is one of the responders for the full key range. Writes are immediately blocked since UT as a responder is unreachable. Reads are still served locally without interruption until ~1.1 secs later, when some healthy server in the cluster raises a heartbeat timeout and initiates a change to a new roster where UT is removed from all responder roles. After waiting 2.6 secs for expiration, normal operations continue.

At ~7.2 secs, we make an explicit roster change request to a server. In contrast to the failure case, this roster change completes in just ~75 ms, which is ~2x cluster-wise RTTs as expected. Impacts on client requests are minor.

Choice of Responders. We run a 10%-write workload on all clients in the WAN setting, while using an increasing set of responders for all keys; WI node is still the leader. We report the cluster-wide average read/write latency and their standard deviation in Figure 17. With more nodes added as responders, read latency tends to zero while write latency increases, revealing the expected tradeoff. This demonstrates the importance of allowing adjustable rosters to help avoid unnecessary taxes on writes.

Coverage of Keys. We repeat the same experiment, but vary the percentage of local-read-enabled keys while fixing the choice of responders to Figure 8(a). The cluster-wide average latencies and their standard deviation across the coverage spectrum are plotted in Figure 18. Results show an expected decrease in read latency and a corresponding increase in write latency as local reads are enabled on more keys. This implies the general strategy of enabling local reads for read-heavy keys while avoiding local reads for write-heavy keys.

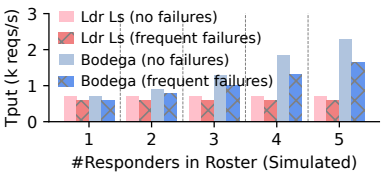


Figure 15. Exaggerated impact of failures *(simulation). See §6.3.*

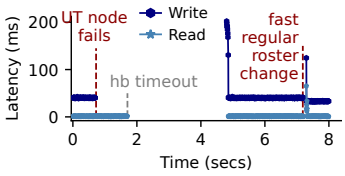


Figure 16. On-failure vs. regular roster changes.

Figure 17. Latency vs. coverage of responders.

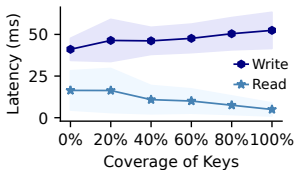
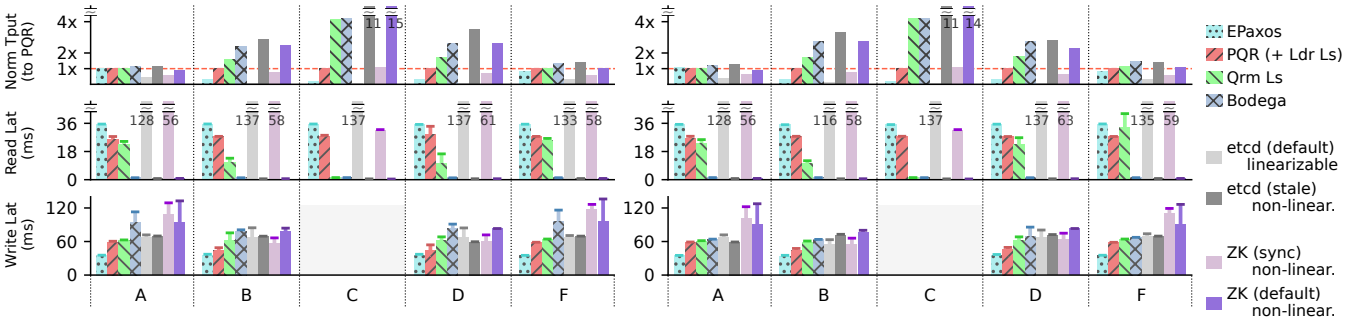


Figure 18. Latency vs. keys covered by roster.



(a) Uniform, full-coverage roster

(b) Zipfian, top-20%-coverage roster

Figure 19. YCSB workloads on Summerset, etcd, & ZooKeeper. *etcd (stale) & ZK (both modes) are not linearizable. See §6.4.*

6.4 Macrobenchmark vs. etcd & ZooKeeper

To evaluate the protocols in a more realistic setup, we compare Summerset protocols with two widely-used coordination services, etcd [35] and ZooKeeper [52], on the WAN cluster. We drive all systems with YCSB [29], the standard KV macrobenchmark. Workloads have the following approximate write ratios (we treat insertions as updates). A: 50% w, B: 5% w, C: 0% w, D: 5% w, F: 25% w.

YCSB Request Distributions. We use 10k keys and construct two scenarios corresponding to two request distributions. ❶ For the Uniform distribution, clients at all locations choose keys uniformly randomly across the key space. Since there are no site-specific preferences for keys, all sites are added as responders for all keys to secure local reads. ❷ For Zipfian, clients at each location choose keys according to a Zipfian-0.99 distribution, skewed towards different sets of keys at different sites; this creates per-site preferences for keys. We then add each site as a responder only for its top-20% accessed keys to derive an asymmetric roster that imposes unnoticeable impacts on write performance.

etcd Modes. We deploy etcd in two modes, both with 120 ms heartbeat intervals. The default mode showcases a standard implementation of vanilla Raft [87]. The *stale* mode turns on the serializable member-local read option for all read requests, delivering sequential consistency by always serving reads locally with past committed values at any server; this represents the ideal upper bound for Bodega.

ZooKeeper Modes. Similarly, we deploy ZooKeeper in two modes, though both are non-linearizable. The default mode is a standard implementation of the ZAB primary-backup protocol [52] that pushes writes to all servers and serves reads locally from anywhere. The *sync* mode is the closest mode to linearizable reads that ZooKeeper clients can get: every read request is preceded by a `sync` API call to force flush all the in-progress writes from the leader to its endpoint server, but all writes that may have completed after the start of the flush are not guaranteed to be seen by the read.

Results. We present the performance results in Figure 19, grouped by workload type and with PQR (+ Ldr Ls) as the normalized throughput baseline. We make the following observations. ❶ Bodega matches (and sometimes surpasses) the performance of sequentially-consistent default ZooKeeper, and is able to keep up with stale etcd across all workloads. This illustrates Bodega's powerful local linearizable read capabilities. The advantage over ZK is due to avoiding Java runtime overheads. ❷ In workload C, both non-linearizable services deliver ~0.3 ms read latency and over 10x throughput gain, while Bodega and Quorum Leases deliver ~1.2 ms latency due to the 1 ms request batching applied. ❸ EPaxos, PQR (+ Ldr Ls), Quorum Leases, and Bodega all show similar patterns coherent to §6.1. With no writes (C), both local read protocols deliver excellent performance. With higher write ratios, Bodega sustains this advantage better than Quorum Leases. ❹ Default etcd and sync ZK have high read latencies of >50 ms because they are classic consensus without leases. ❺ Comparing the Uniform scenario with Zipfian, the only notable difference is that Bodega exhibits higher write latencies close to ZK in Uniform. This is expected because Bodega writes need to reach all nodes as they are all responders.

7 Discussion

We discuss topics that are out of the scope of this paper but are interesting directions for future work. We also clarify non-goals of Bodega.

Partial Network Partitioning. Using heartbeat timeout-based failure detection for leader step-up is known to risk liveness under partial network partitioning [86], and the same holds true for roster lease activations. Common techniques such as pre-votes [86] and transparent re-routing [2] can be deployed to easily eliminate this issue.

Generalization of Roster Leases. We observe that the activation procedure of roster leases shares similarities with *broadcast-based* (randomized) consensus [16, 83, 89]. It is a practical application of all-to-all broadcast in a non-adversarial setting for one-off agreements on the roster ballot. Combined with leases, this technique can be used to establish fault-tolerant agreement on any general "roster metadata" that change infrequently, not limited to leadership and assignment of responders as in Bodega. Possible extensions may include cluster membership, asymmetric quorum sizes, and node-specific performance and reliability hints.

Bounded Staleness Support. With simple modifications, Bodega can extend beyond linearizability and support fast local reads that can tolerate (but require) bounded staleness measured in the maximum version difference with the latest committed write. When a non-leader responder receives a read that allows up to x versions stale, it can traverse the tail of its log in reverse and search for at most x occurrences of the key, returning the latest committed version if found.

Non-Goals of Bodega. Bodega is primarily designed for wide-area linearizable read workloads; its improvements become less pronounced in other consensus use cases outside of this regime. These include: ❶ intra-datacenter replication where locality isn't significant, ❷ write-dominant workloads (see Figure 13 showing diminishing throughput improvement at 50% writes), and ❸ weaker consistency models where arbitrarily stale reads or inconsistent reads are tolerable. They may still leverage Bodega's roster leases as a mechanism to share cluster-wise information at negligible overhead (see the paragraph on generalization above), but would not observe the most immediate gains from localized linearizable reads.

8 Related Work

We list additional notable related work in this section.

Consensus & Read Optimizations. §2.3 and §4.1 have covered in detail the most essential related work, including classic consensus algorithms [60, 61, 65, 85–87], leaderless or multi-leader approaches [1, 27, 34, 56, 62, 63, 76, 80, 98, 104, 106], and read leases [10, 25, 41, 81, 82, 105].

Flexible quorum sizes have been discussed in classic literature [46] as well as in recent proposals such as Flexible Paxos [42, 48, 84, 111], where quorum shapes other than majority are explored to establish asymmetric performance between commands; Bodega is a modern manifestation of fault-tolerant, asymmetric read/write quorums leveraging leases. Optimistic holding shares similarity to wait-vs.-abort in database concurrency control [44, 58] and spin-then-park mutex locks [55]; we wait when waiting is expected to be faster.

Shared Logs & Lazy Ordering for Writes. Shared logs are a common abstraction found in cloud systems and are usually backed by primary-backup-style protocols [11–14, 20, 31, 73]. CAD [38], Skyros [39], and LazyLog [74] are a series of work on a *lazy ordering* optimization for writes and shared log appends. It hides a significant portion of write latency but could hurt read performance in contended cases.

Synchronized Clocks. Recent works demonstrate production-ready implementations of synchronized clocks [30, 70] and designs that take advantage of them through timestamp heuristics [32, 40]. Chandra et al. presented a formal, optimal lease algorithm that assumes synchronized clocks [18, 26].

Weaker Consistency Models. While this paper focuses on linearizable solutions, services may choose to provide weaker consistency models, which are less intuitive to reason about and harder to program against, but usually offer better performance. Common choices include sequential consistency that allows time-traveling reads [6, 45, 59], causal consistency that focuses on client session-local properties [9, 72, 78], and eventual consistency with bounded staleness [92, 109]. There are also studies that explore supporting a mixture of consistency models in one system [75, 113].

9 Conclusion

We present Bodega, a wide-area consensus protocol that enables localized linearizable reads at anywhere, anytime. Bodega achieves this via all-to-all roster leases off the critical path to establish responder assignment without compromising fault tolerance, delivering extreme read performance comparable to sequentially-consistent production systems. We believe Bodega is a valuable step towards performance-optimal wide-area replication for critical workloads of the modern cloud.

References

[1] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tevfik Kosar. 2020. WPaxos: Wide Area Network Flexible Consensus. *IEEE Trans. Parallel Distrib. Syst.* 31, 1 (Jan. 2020), 211–223. doi:10.1109/TPDS.2019.2929793

[2] Mohammed Alfatafta, Basil Alkhatib, Ahmed Alquraan, and Samer Al-Kiswany. 2020. Toward a Generic Fault Tolerance Technique for Partial Network Partitioning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 351–368. https://www.usenix.org/conference/osdi20/presentation/alfatafta

[3] Amazon Web Services, Inc. 2021. *Availability and Beyond: Understanding and Improving the Resilience of Distributed Systems on AWS*. Whitepaper AWS Whitepaper. Amazon Web Services, Inc. https://docs.aws.amazon.com/pdfs/whitepapers/latest/availability-and-beyond-improving-resilience/availability-and-beyond-improving-resilience.pdf Accessed: 2025-12-10.

[4] Artem on StackOverflow. 2017. Is ZooKeeper always consistent in terms of CAP theorem? https://stackoverflow.com/questions/35387774. Accessed: 2024-12-01.

[5] Balaji Arun, Sebastiano Peluso, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. 2017. Speeding up Consensus by Chasing Fast Decisions. In *47th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 49–60. doi:10.1109/DSN.2017.35

[6] Hagit Attiya and Jennifer L. Welch. 1994. Sequential Consistency versus Linearizability. *ACM Trans. Comput. Syst.* 12, 2 (may 1994), 91–122. doi:10.1145/176575.176576

[7] AWS. 2024. AWS Global Infrastructure. https://aws.amazon.com/about-aws/global-infrastructure/, Last accessed on 2024-04-28.

[8] AWS. 2024. Workload Characteristics. https://docs.aws.amazon.com/prescriptive-guidance/latest/oracle-exadata-blueprint/workload-characteristics.html. Accessed: 2024-12-01.

[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* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 761–772. doi:10.1145/2463676.2465279

[10] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*. 223–234. http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper32.pdf

[11] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, Jingming Liu, Filip Gruszczynski, Xianan Zhang, Huy Hoang, Ahmed Yossef, Francois Richard, and Yee Jiun Song. 2020. Virtual Consensus in Delos. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 617–632. https://www.usenix.org/conference/osdi20/presentation/balakrishnan

[12] Mahesh Balakrishnan, Dahlia Malkhi, John D. Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. 2013. CORFU: A distributed shared log. *ACM Trans. Comput. Syst.* 31, 4, Article 10 (Dec. 2013), 24 pages. doi:10.1145/2535930

[13] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. 2013. Tango: Distributed Data Structures over a Shared Log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 325–340. doi:10.1145/2517349.2522732

[14] Mahesh Balakrishnan, Chen Shen, Ahmed Jafri, Suyog Mapara, David Geraghty, Jason Flinn, Vidhya Venkat, Ivailo Nedelchev, Santosh Ghosh, Mihir Dharamshi, Jingming Liu, Filip Gruszczynski, Jun Li, Rounak Tibrewal, Ali Zaveri, Rajeev Nagar, Ahmed Yossef, Francois Richard, and Yee Jiun Song. 2021. Log-structured Protocols in Delos. In *Proceedings of the ACM 28th Symposium on Operating Systems Principles* (Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 538–552. doi:10.1145/3477132.3483544

[15] Jeff Barr. 2023. Amazon S3 Update – Strong Read-After-Write Consistency. https://aws.amazon.com/blogs/aws/amazon-s3-update-strong-read-after-write-consistency/, Last accessed on 2023-11-19.

[16] Michael Ben-Or. 1983. Another advantage of free choice (Extended Abstract): Completely asynchronous agreement protocols. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing* (Montreal, Quebec, Canada) *(PODC '83)*. Association for Computing Machinery, New York, NY, USA, 27–30. doi:10.1145/800221.806707

[17] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data* (San Jose, California, USA) *(SIGMOD '95)*. Association for Computing Machinery, New York, NY, USA, 1–10. doi:10.1145/223784.223785

[18] Changyu Bi, Vassos Hadzilacos, and Sam Toueg. 2022. Parameterized algorithm for replicated objects with local reads. arXiv:2204.01228 [cs.DC] https://arxiv.org/abs/2204.01228

[19] Marc Brooker, Tao Chen, and Fan Ping. 2020. Millions of tiny databases. In *NSDI 2020*. https://www.amazon.science/publications/millions-of-tiny-databases

[20] 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

[21] Mike Burrows. 2006. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, Washington) *(OSDI '06)*. USENIX Association, USA, 335–350.

[22] Georgia Butler. 2024. Google Cloud accidentally deleted UniSuper's Private Cloud Subscription. *Data Center Dynamics* (2024). https://www.datacenterdynamics.com/en/news/google-cloud-accidentally-deleted-unisupers-private-cloud-subscription/

[23] Miguel Castro and Barbara Liskov. 2002. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.* 20, 4 (Nov. 2002), 398–461. doi:10.1145/571637.571640

[24] Data Centre. 2024. Alibaba Cloud hit by Digital Realty fire in Singapore. *Frontier Enterprise* (2024). https://www.frontier-enterprise.com/alibaba-cloud-hit-by-digital-realty-fire-in-singapore/

[25] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos Made Live: An Engineering Perspective. In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing* (Portland, OR, USA) *(PODC '07)*. Association for Computing Machinery, New York, NY, USA, 398–407. doi:10.1145/1281100.1281103

[26] Tushar D. Chandra, Vassos Hadzilacos, and Sam Toueg. 2016. An Algorithm for Replicated Objects with Efficient Reads. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing* (Chicago, Illinois, USA) *(PODC '16)*. Association for Computing Machinery, New York, NY, USA, 325–334. doi:10.1145/2933057.2933111

[27] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. 2019. Linearizable Quorum Reads in Paxos. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*. USENIX Association, Renton, WA. https://www.usenix.org/conference/hotstorage19/presentation/charapko

[28] Google Cloud. 2024. Google Cloud locations. https://cloud.google.com/about/locations/, Last accessed on 2024-11-30.

[29] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, IN, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. doi:10.1145/1807128.1807152

[30] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3, Article 8 (aug 2013), 22 pages. doi:10.1145/2491245

[31] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert Van Renesse. 2020. Scalog: Seamless Reconfiguration and Total Order in a Scalable Shared Log. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 325–338. https://www.usenix.org/conference/nsdi20/presentation/ding

[32] Jiaqing Du, Daniele Sciascia, Sameh Elnikety, Willy Zwaenepoel, and Fernando Pedone. 2014. Clock-RSM: Low-Latency Inter-datacenter State Machine Replication Using Loosely Synchronized Physical Clocks. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 343–354. doi:10.1109/DSN.2014.42

[33] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference*. 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19

[34] Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. 2020. State-Machine Replication for Planet-Scale Systems. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 24, 15 pages. doi:10.1145/3342195.3387543

[35] etcd. 2023. etcd: A distributed, reliable key-value store for the most critical data. https://etcd.io/, Last accessed on 2023-11-13.

[36] FireScroll. 2023. FireScroll: The config database to deploy everywhere. https://github.com/FireScroll/FireScroll, Last accessed on 2024-09-05.

[37] Pedro Fouto, Nuno Preguiça, and Joao Leitão. 2022. High Throughput Replication with Integrated Membership Management. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 575–592. https://www.usenix.org/conference/atc22/presentation/fouto

[38] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2020. Strong and Efficient Consistency with Consistency-Aware Durability. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 323–337. https://www.usenix.org/conference/fast20/presentation/ganesan

[39] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2021. Exploiting Nil-Externality for Fast Replicated Storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 440–456. doi:10.1145/3477132.3483543

[40] Jinkun Geng, Anirudh Sivaraman, Balaji Prabhakar, and Mendel Rosenblum. 2022. Nezha: Deployable and High-Performance Consensus Using Synchronized Clocks. *Proc. VLDB Endow.* 16, 4 (dec 2022), 629–642. doi:10.14778/3574245.3574250

[41] C. Gray and D. Cheriton. 1989. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles (SOSP '89)*. Association for Computing Machinery, New York, NY, USA, 202–210. doi:10.1145/74850.74870

[42] Joshua Guarnieri and Aleksey Charapko. 2023. Linearizable Low-latency Reads at the Edge. In *Proceedings of the 10th Workshop on Principles and Practice of Consistency for Distributed Data* (Rome, Italy) *(PaPoC '23)*. Association for Computing Machinery, New York, NY, USA, 77–83. doi:10.1145/3578358.3591327

[43] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. 2014. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SOCC '14)*. Association for Computing Machinery, New York, NY, USA, 1–14. doi:10.1145/2670979.2670986

[44] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. 2017. An evaluation of distributed concurrency control. *Proc. VLDB Endow.* 10, 5 (Jan 2017), 553–564. doi:10.14778/3055540.3055548

[45] Jeffrey Helt, Matthew Burke, Amit Levy, and Wyatt Lloyd. 2021. Regular Sequential Serializability and Regular Sequential Consistency. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 163–179. doi:10.1145/3477132.3483566

[46] Maurice Herlihy. 1987. Dynamic quorum adjustment for partitioned data. *ACM Trans. Database Syst.* 12, 2 (Jun 1987), 170–194. doi:10.1145/22952.22953

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

[48] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. 2016. Flexible Paxos: Quorum intersection revisited. arXiv:1608.06696 [cs.DC]

[49] Guanzhou Hu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2024. A Unified, Practical, and Understandable Summary of Non-transactional Consistency Levels in Distributed Replication. arXiv:2409.01576 [cs.DC] https://arxiv.org/abs/2409.01576

[50] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. 2020. TiDB: A Raft-Based HTAP Database. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3072–3084. doi:10.14778/3415478.3415535

[51] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. 2017. Gray Failure: The Achilles' Heel of Cloud-Scale Systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (Whistler, BC, Canada) *(HotOS '17)*. Association for Computing Machinery, New York, NY, USA, 150–155. doi:10.1145/3102980.3103005

[52] 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* (Boston, MA) *(USENIXATC'10)*. USENIX Association, USA, 11.

[53] Randall Hunt. 2017. Keeping Time With Amazon Time Sync Service. https://aws.amazon.com/blogs/aws/keeping-time-with-amazon-time-sync-service/. Accessed: 2017-11-29.

[54] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. 2017. Canopy: An End-to-End Performance Tracing And Analysis System. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 34–50. doi:10.1145/3132747.3132749

[55] Anna R. Karlin, Kai Li, Mark S. Manasse, and Susan Owicki. 1991. Empirical studies of competitve spinning for a shared-memory multiprocessor. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles* (Pacific Grove, California, USA) *(SOSP '91)*. Association for Computing Machinery, New York, NY, USA, 41–55. doi:10.1145/121132.286599

[56] Antonios Katsarakis, Vasilis Gavrielatos, M.R. Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. 2020. Hermes: A Fast, Fault-Tolerant and Linearizable Replication Protocol. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 201–217. doi:10.1145/3373376.3378496

[57] KRaft. 2025. KRaft: Apache Kafka Without ZooKeeper. https://developer.confluent.io/learn/kraft/, Last accessed on 2025-04-12.

[58] H. T. Kung and John T. Robinson. 1981. On optimistic methods for concurrency control. *ACM Trans. Database Syst.* 6, 2 (jun 1981), 213–226. doi:10.1145/319566.319567

[59] Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers C-28* 9, 9 (September 1979), 690–691. doi:10.1109/TC.1979.1675439

[60] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (may 1998), 133–169. doi:10.1145/279227.279229

[61] Leslie Lamport. 2001. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (December 2001), 51–58. https://www.microsoft.com/en-us/research/publication/paxos-made-simple/

[62] Leslie Lamport. 2005. Generalized consensus and Paxos. *Microsoft Research Technical Report* (2005).

[63] Leslie Lamport. 2006. Fast Paxos. *Distrib. Comput.* 19, 2 (oct 2006), 79–103. doi:10.1007/s00446-006-0005-x

[64] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. 2009. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing* (Calgary, AB, Canada) *(PODC '09)*. Association for Computing Machinery, New York, NY, USA, 312–313. doi:10.1145/1582716.1582783

[65] Butler Lampson. 2001. The ABCD's of Paxos. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing* (Newport, RI, USA) *(PODC '01)*. Association for Computing Machinery, New York, NY, USA, 13. doi:10.1145/383962.383969

[66] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John Ousterhout. 2015. Implementing Linearizability at Large Scale and Low Latency. In *Proceedings of the 25th Symposium on Operating Systems Principles* (CA) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 71–86. doi:10.1145/2815400.2815416

[67] Edward K. F. Lee and Chandramohan A. Thekkath. 1996. Petal: distributed virtual disks. In *ASPLOS VII (ASPLOS96)*. ACM, 84–92. doi:10.1145/237090.237157

[68] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. 2016. Globally Synchronized Time via Datacenter Networks. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 454–467. doi:10.1145/2934872.2934885

[69] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. 2016. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 467–483. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/li

[70] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, and Amin Vahdat. 2020. Sundial: Fault-tolerant Clock Synchronization for Datacenters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 1171–1186. https://www.usenix.org/conference/osdi20/presentation/li-yuliang

[71] Linux man pages. 2011. tc-netem(8) — Linux manual page. https://man7.org/linux/man-pages/man8/tc-netem.8.html. [Online; accessed 29-November-2023].

[72] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 401–416. doi:10.1145/2043556.2043593

[73] Joshua Lockerman, Jose M. Faleiro, Juno Kim, Soham Sankaran, Daniel J. Abadi, James Aspnes, Siddhartha Sen, and Mahesh Balakrishnan. 2018. The FuzzyLog: A Partially Ordered Shared Log. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 357–372. https://www.usenix.org/conference/osdi18/presentation/lockerman

[74] Xuhao Luo, Shreesha G. Bhat, Jiyu Hu, Ramnatthan Alagappan, and Aishwarya Ganesan. 2024. LazyLog: A New Shared Log Abstraction for Low-Latency Applications. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 296–312. doi:10.1145/3694715.3695983

[75] Kai Ma, Cheng Li, Enzuo Zhu, Ruichuan Chen, Feng Yan, and Kang Chen. 2024. Noctua: Towards Automated and Practical Fine-grained Consistency Analysis. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 704–719. doi:10.1145/3627703.3629570

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

[77] Jim Martin, Jack Burbank, William Kasch, and Professor David L. Mills. 2010. Network Time Protocol Version 4: Protocol and Algorithms Specification. RFC 5905. doi:10.17487/RFC5905

[78] 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 *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 453–468. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/mehdi

[79] Microsoft. 2024. Azure Global Infrastructure. https://datacenters.microsoft.com/globe/explore/, Last accessed on 2024-11-30.

[80] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 358–372. doi:10.1145/2517349.2517350

[81] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2014. Paxos Quorum Leases: Fast Reads Without Sacrificing Writes. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SOCC '14)*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/2670979.2671001

[82] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2014. Paxos Quorum Leases: Fast Reads Without Sacrificing Writes. *Carnegie Mellon University PDL Technical Report* (2014).

[83] Achour Mostefaoui, Matthieu Perrin, and Julien Weibel. 2024. Brief Announcement: Randomized Consensus: Common Coins Are not the Holy Grail!. In *Proceedings of the 43rd ACM Symposium on Principles of Distributed Computing* (Nantes, France) *(PODC '24)*. Association for Computing Machinery, New York, NY, USA, 36–39. doi:10.1145/3662158.3662824

[84] Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2018. DPaxos: Managing Data Closer to Users for Low-Latency and Mobile Applications. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1221–1236. doi:10.1145/3183713.3196928

[85] Brian M. Oki and Barbara H. Liskov. 1988. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing* (Toronto, Ontario, Canada) *(PODC '88)*. Association for Computing Machinery, New York, NY, USA, 8–17. doi:10.1145/62546.62549

[86] Diego Ongaro. 2014. *Consensus: Bridging Theory and Practice*. Ph. D. Dissertation. Stanford, CA, USA. Advisor(s) K., Ousterhout, John and David, Mazières, and Mendel, Rosenblum,. AAI28121474.

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

[88] Team Live Optics. 2021. Live Optics Basics: Read / Write Ratio. https://support.liveoptics.com/hc/en-us/articles/229590547-Live-Optics-Basics-Read-Write-Ratio. Accessed: 2024-12-01.

[89] Haochen Pan, Jesse Tuglu, Neo Zhou, Tianshu Wang, Yicheng Shen, Xiong Zheng, Joseph Tassarotti, Lewis Tseng, and Roberto Palmieri. 2021. Rabia: Simplifying State-Machine Replication Through Randomization. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 472–487. doi:10.1145/3477132.3483582

[90] Seo Jin Park and John Ousterhout. 2019. Exploiting Commutativity For Practical Fast Replication. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 47–64. https://www.usenix.org/conference/nsdi19/presentation/park

[91] Suraj Pasuparthy and Lokesh Agarwal. 2023. Benchmarking Spanner's price-performance for key-value workloads. https://cloud.google.com/blog/products/databases/benchmarking-spanner-for-key-value-workloads/. Accessed: 2024-12-01.

[92] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. 1997. Flexible update propagation for weakly consistent replication. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (Saint Malo, France) *(SOSP '97)*. Association for Computing Machinery, New York, NY, USA, 288–301. doi:10.1145/268998.266711

[93] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. 2015. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 43–57. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/ports

[94] RabbitMQ. 2025. RabbitMQ: One broker to queue them all. https://www.rabbitmq.com/, Last accessed on 2025-04-08.

[95] Redpanda. 2024. Redpanda: The Unified Streaming Data Platform. https://www.redpanda.com/, Last accessed on 2024-09-05.

[96] Robbert Van Renesse and Fred B. Schneider. 2004. Chain Replication for Supporting High Throughput and Availability. In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*. USENIX Association, CA. https://www.usenix.org/conference/osdi-04/chain-replication-supporting-high-throughput-and-availability

[97] David K. Rensin. 2015. *Kubernetes - Scheduling the Future at Cloud Scale*. O'Reilly and Associates, 1005 Gravenstein Highway North Sebastopol, CA 95472. All pages. http://www.oreilly.com/webops-perf/free/kubernetes.csp

[98] Fedor Ryabinin, Alexey Gotsman, and Pierre Sutra. 2024. SwiftPaxos: Fast Geo-Replicated State Machines. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 345–369. https://www.usenix.org/conference/nsdi24/presentation/ryabinin

[99] Fred B. Schneider. 1990. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.* 22, 4 (dec 1990), 299–319. doi:10.1145/98163.98167

[100] ScyllaDB. 2023. Beyond Legacy NoSQL: 7 Design Principles Behind ScyllaDB. https://lp.scylladb.com/real-time-big-data-database-principles-thanks.html, Last accessed on 2023-11-13.

[101] Hatem Takruri, Ibrahim Kettaneh, Ahmed Alquraan, and Samer Al-Kiswany. 2020. FLAIR: Accelerating Reads with Consistency-Aware Network Routing. In *17th USENIX Symposium on Networked Systems Design & Implementation (NSDI 20)*. USENIX Association, CA, 723–737. https://usenix.org/conference/nsdi20/presentation/takruri

[102] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. 1994. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the Third International Conference on on Parallel and Distributed Information Systems* (Autin, Texas, USA) *(PDIS '94)*. IEEE Computer Society Press, Washington, DC, USA, 140–150. doi:10.1109/PDIS.1994.331722

[103] TigerBeetle. 2024. TigerBeetle: The Financial Transactions Database. https://tigerbeetle.com/, Last accessed on 2024-11-12.

[104] Sarah Tollman, Seo Jin Park, and John Ousterhout. 2021. EPaxos Revisited. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 613–632. https://www.usenix.org/conference/nsdi21/presentation/tollman

[105] Bohdan Trach, Rasha Faqeh, Oleksii Oleksenko, Wojciech Ozga, Pramod Bhatotia, and Christof Fetzer. 2020. T-Lease: a trusted lease primitive for distributed systems. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 387–400. doi:10.1145/3419111.3421273

[106] Muhammed Uluyol, Anthony Huang, Ayush Goel, Mosharaf Chowdhury, and Harsha V. Madhyastha. 2020. Near-Optimal Latency Versus Cost Tradeoffs in Geo-Distributed Storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 157–180. https://www.usenix.org/conference/nsdi20/presentation/uluyol

[107] Nathan VanBenschoten, Arul Ajmani, Marcus Gartner, Andrei Matei, Aayush Shah, Irfan Sharif, Alexander Shraer, Adam Storm, Rebecca Taft, Oliver Tan, Andy Woods, and Peyton Walters. 2022. Enabling the Next Generation of Multi-Region Applications with CockroachDB. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2312–2325. doi:10.1145/3514221.3526053

[108] Kaushik Veeraraghavan, Justin Meza, Scott Michelson, Sankaralingam Panneerselvam, Alex Gyori, David Chou, Sonia Margulis, Daniel Obenshain, Shruti Padmanabha, Ashish Shah, Yee Jiun Song, and Tianyin Xu. 2018. Maelstrom: Mitigating Datacenter-level Disasters by Draining Interdependent Traffic Safely and Efficiently. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 373–389. https://www.usenix.org/conference/osdi18/presentation/veeraraghavan

[109] Werner Vogels. 2008. Eventually Consistent: Building Reliable Distributed Systems at a Worldwide Scale Demands Trade-Offs Between Consistency and Availability. *Queue* 6, 6 (oct 2008), 14–19. doi:10.1145/1466443.1466448

[110] Zhaoguo Wang, Changgeng Zhao, Shuai Mu, Haibo Chen, and Jinyang Li. 2019. On the Parallels between Paxos and Raft, and how to Port Optimizations. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (Toronto ON, Canada) *(PODC '19)*. Association for Computing Machinery, New York, NY, USA, 445–454. doi:10.1145/3293611.3331595

[111] Michael Whittaker, Aleksey Charapko, Joseph M. Hellerstein, Heidi Howard, and Ion Stoica. 2021. Read-Write Quorum Systems Made Practical. In *Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data* (Online, United Kingdom) *(PaPoC '21)*. Association for Computing Machinery, New York, NY, USA, Article 7, 8 pages. doi:10.1145/3447865.3457962

[112] Jian Yi, Qing Li, Bin Zhang, Yong Jiang, Dan Zhao, Yuan Yang, and Zhenhui Yuan. 2023. Gleaning the Consensus for Linearizable and Conflict-Free Per-Replica Local Reads. In *Proceedings of the 7th Asia-Pacific Workshop on Networking* (Hong Kong, China) *(APNet '23)*. Association for Computing Machinery, New York, NY, USA, 143–149. doi:10.1145/3600061.3603175

[113] Haifeng Yu and Amin Vahdat. 2000. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings*

of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4 (San Diego, California) *(OSDI'00)*. USENIX Association, USA, Article 21.

[114] Hanyu Zhao, Quanlu Zhang, Zhi Yang, Ming Wu, and Yafei Dai. 2018. SDPaxos: Building Efficient Semi-Decentralized Geo-replicated State Machines. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 68–81. doi:10.1145/3267809.3267837

A Formal TLA$^+$ Specification of Bodega

We also present a formal specification of Bodega written as a PlusCal algorithm that can be auto-translated into TLA$^+$. The specification builds on top of MultiPaxos and introduces all-to-all roster lease actions to support local reads, faithful to the presentation in Figure 7. Please refer to the inlined comments for specification details.

The specification has been model-checked for both *linearizability* and *fault-tolerance* properties on symbolic inputs of 3 nodes (thus 1 allowed failure), 3 ballot numbers, 2 distinct write requests plus 2 distinct read requests, 3 lease time ticks, and all possible choices of responders. These inputs should be large enough to explore the distinct, interesting execution paths of the protocol. Model checking took 43 hours on a single machine with 96 cores and 768GiB memory, and generated 4,274,883,464 distinct states.

B Artifact Evaluation Instructions

Bodega's source code (as part of the Summerset key-value store) is publicly available on GitHub at https://github.com/josehu07/summerset/tree/bodega-artifact.

Artifact evaluation instructions can be found at https://github.com/josehu07/summerset/blob/bodega-artifact/publish/bodega/ARTIFACT.md. The document contains detailed instructions on setting up a distributed consensus system environment and reproducing all the figures presented in this paper.

MODULE *Bodega*

EXTENDS *FiniteSets*, *Sequences*, *Integers*, *TLC*

Model inputs & assumptions.

CONSTANT *Replicas*, *symmetric set of server nodes*
Writes, *symmetric set of write commands(each w/unique value)*
Reads, *symmetric set of read commands*
MaxBallot, *maximum ballot pickable for leader preemption*
TGuard, *lease guard phase window length(in abstract ticks)*
TLease, *lease renewal extend window length(in abstract ticks)*
MaxTime, *upper bound on abstract time for model checking*
NodeFailuresOn *if true, turn on node failures injection*

$$ReplicasAssumption \triangleq \land IsFiniteSet(Replicas) \land Cardinality(Replicas) \geq 1 \land \text{“none”} \notin Replicas$$

$$Population \triangleq Cardinality(Replicas)$$

$$MajorityNum \triangleq (Population \div 2) + 1$$

$$WritesAssumption \triangleq \land IsFiniteSet(Writes) \land Cardinality(Writes) \geq 1 \land \text{“nil”} \notin Writes$$

$$ReadsAssumption \triangleq \land IsFiniteSet(Reads) \land Cardinality(Reads) \geq 0 \land \text{“nil”} \notin Writes$$

$$MaxBallotAssumption \triangleq \land MaxBallot \in Nat \land MaxBallot \geq 2$$

$$TGuardAssumption \triangleq \land TGuard \in Nat \land TGuard \geq 1$$

$$TLeaseAssumption \triangleq \land TLease \in Nat \land TLease \geq 1$$

$$MaxTimeAssumption \triangleq \land MaxTime \in Nat \land MaxTime \geq TGuard + TLease$$

$$NodeFailuresOnAssumption \triangleq NodeFailuresOn \in \text{BOOLEAN}$$

ASSUME $\land ReplicasAssumption \land WritesAssumption \land ReadsAssumption \land MaxBallotAssumption \land TGuardAssumption \land TLeaseAssumption \land MaxTimeAssumption \land NodeFailuresOnAssumption$

Useful constants & typedefs.

$$Commands \triangleq Writes \cup Reads$$

$$NumWrites \triangleq Cardinality(Writes)$$

$NumReads \triangleq Cardinality(Reads)$

$NumCommands \triangleq Cardinality(Commands)$

$Range(seq) \triangleq \{seq[i] : i \in 1 \,..\, Len(seq)\}$

Client observable events.

$$\begin{aligned} ClientEvents \triangleq\ & [type : \{\text{“Req”}\},\ cmd : Commands] \\ \cup\ & [type : \{\text{“Ack”}\},\ cmd : Commands, \\ & \qquad val\ : \{\text{“nil”}\} \cup Writes, \\ & \qquad by : Replicas] \end{aligned}$$

$ReqEvent(c) \triangleq [type \mapsto \text{“Req”},\ cmd \mapsto c]$

$AckEvent(c, v, n) \triangleq [type \mapsto \text{“Ack”},\ cmd \mapsto c,\ val \mapsto v,\ by \mapsto n]$

val is the old value for a write command

$$\begin{aligned} InitPending \triangleq\ & (\text{CHOOSE } ws \in [1 \,..\, Cardinality(Writes) \to Writes] \\ & \qquad : Range(ws) = Writes) \\ \circ\ & (\text{CHOOSE } rs \in [1 \,..\, Cardinality(Reads) \to Reads] \\ & \qquad : Range(rs) = Reads) \end{aligned}$$

Server – side consensus constants & states.

$Ballots \triangleq 1 \,..\, MaxBallot$

$Slots \triangleq 1 \,..\, NumWrites$

$InfinitySlot \triangleq NumWrites + 1$

sentinel “infinitely-high” *slot, used as commitPrev s*

safe “haven’t learned yet” marker

$$\begin{aligned} Rosters \triangleq\ & \{ros \in [bal : Ballots, \\ & \qquad leader : Replicas, \\ & \qquad responders : \text{SUBSET } Replicas] : \\ & ros.leader \notin ros.responders\} \end{aligned}$$

for a smaller state space we exclude leader from the literal set of *responders* (but always treating it as a responder)

$Roster(b, l, resps) \triangleq [bal \mapsto b,\ leader \mapsto l,\ responders \mapsto resps]$

each new ballot number maps to a new roster; this includes the change of leader (as in classic *MultiPaxos*) and/or the change of who’re *responders*

$NullRoster \triangleq [bal \mapsto 0,\ leader \mapsto \text{“none”},\ responders \mapsto \{\}]$

$Statuses \triangleq \{\text{“Preparing”}, \text{“Accepting”}, \text{“Committed”}\}$

$$\begin{aligned} InstStates \triangleq\ & [status : \{\text{“Empty”}\} \cup Statuses, \\ & \quad write : \{\text{“nil”}\} \cup Writes, \\ & \quad voted : [bal : \{0\} \cup Ballots, \\ & \qquad write : \{\text{“nil”}\} \cup Writes]] \end{aligned}$$

$$\begin{aligned} NullInst \triangleq\ & [status \mapsto \text{“Empty”}, \\ & \quad write \mapsto \text{“nil”}, \\ & \quad voted \mapsto [bal \mapsto 0,\ write \mapsto \text{“nil”}]] \end{aligned}$$

Lease-side constants & typedefs.

$Times \triangleq 1 \,..\, MaxTime$

$ExpireTimes \triangleq 0 \,..\, (MaxTime + TGuard + TLease)$

stored expiration / guard deadlines; 0 is the "null" time

$SeqNums \triangleq Nat$

per-pair monotone *seq* nums on lease messages for dedup

$GrantorStatuses \triangleq \{\text{“None”}, \text{“Guarding”}, \text{“Renewing”}, \text{“Revoking”}\}$
$GranteeStatuses \triangleq \{\text{“None”}, \text{“Guarded”}, \text{“Renewed”}\}$

$$\begin{aligned} GrantorState \triangleq [&status : GrantorStatuses,\\ &guardExpire : ExpireTimes,\\ &leaseExpire\ \ : ExpireTimes,\\ &ros : \{NullRoster\} \cup Rosters,\\ &seq : SeqNums] \end{aligned}$$

$$\begin{aligned} GranteeState \triangleq [&status : GranteeStatuses,\\ &guardExpire : ExpireTimes,\\ &leaseExpire\ \ : ExpireTimes,\\ &ros : \{NullRoster\} \cup Rosters,\\ &seq : SeqNums] \end{aligned}$$

$$\begin{aligned} NullGrantorState \triangleq [&status \mapsto \text{“None”},\\ &guardExpire \mapsto 0,\\ &leaseExpire\ \ \mapsto 0,\\ &ros \mapsto NullRoster,\\ &seq \mapsto 0] \end{aligned}$$

$$\begin{aligned} NullGranteeState \triangleq [&status \mapsto \text{“None”},\\ &guardExpire \mapsto 0,\\ &leaseExpire\ \ \mapsto 0,\\ &ros \mapsto NullRoster,\\ &seq \mapsto 0] \end{aligned}$$

Merged per-node state: consensus *fields* + lease *fields*.

$$\begin{aligned} NodeStates \triangleq [&leader : \{\text{“none”}\} \cup Replicas,\\ &commitUpTo : \{0\} \cup Slots,\\ &commitPrev\ \ : \{0\} \cup Slots \cup \{InfinitySlot\},\\ &balPrepared\ \ \ \ : \{0\} \cup Ballots,\\ &balMaxKnown : \{0\} \cup Ballots,\\ &rosMaxKnown : \{NullRoster\} \cup Rosters,\\ &insts : [Slots \to InstStates],\\ &asGrantor : [Replicas \to GrantorState],\\ &asGrantee : [Replicas \to GranteeState]] \end{aligned}$$

$$\begin{aligned} NullNode \triangleq [&leader \mapsto \text{“none”},\\ &commitUpTo \mapsto 0,\\ &commitPrev\ \ \mapsto 0,\\ &balPrepared\ \ \ \ \mapsto 0,\\ &balMaxKnown \mapsto 0,\\ &rosMaxKnown \mapsto NullRoster,\\ &insts \mapsto [s \in Slots \mapsto NullInst],\\ &asGrantor \mapsto [p \in Replicas \mapsto NullGrantorState],\\ &asGrantee \mapsto [f \in Replicas \mapsto NullGranteeState]] \end{aligned}$$

commitPrev is the last slot which might have been committed by an old leader; a newly joined node can safely serve reads locally only after its log has been committed up to this slot

rosMaxKnown is the roster that came with *balMaxKnown*

$asGrantor[p]$ is the grantor-side lease state & timers toward grantee p; $asGrantee[f]$ is the grantee-side state for grantor f

$$
\begin{array}{l}
FirstEmptySlot(insts) \triangleq \\
\quad \text{IF } \forall\, s \in Slots : insts[s].status \neq \text{“Empty”} \\
\qquad \text{THEN } InfinitySlot \\
\qquad \text{ELSE CHOOSE } s \in Slots : \\
\qquad\qquad \wedge\, insts[s].status = \text{“Empty”} \\
\qquad\qquad \wedge\, \forall\, t \in 1 \,..\, (s-1) : insts[t].status \neq \text{“Empty”}
\end{array}
$$

$$
\begin{array}{l}
LastNonEmptySlot(insts) \triangleq \\
\quad \text{IF } \forall\, s \in Slots : insts[s].status = \text{“Empty”} \\
\qquad \text{THEN } 0 \\
\qquad \text{ELSE CHOOSE } s \in Slots : \\
\qquad\qquad \wedge\, insts[s].status \neq \text{“Empty”} \\
\qquad\qquad \wedge\, \forall\, t \in (s+1) \,..\, NumWrites : insts[t].status = \text{“Empty”}
\end{array}
$$

Service-internal consensus messages.

$$PrepareMsgs \triangleq [type : \{\text{“Prepare”}\}, src : Replicas, bal : Ballots]$$

$$PrepareMsg(r, b) \triangleq [type \mapsto \text{“Prepare”}, src \mapsto r, bal \mapsto b]$$

$$InstsVotes \triangleq [Slots \to [bal : \{0\} \cup Ballots, write : \{\text{“nil”}\} \cup Writes]]$$

$$VotesByNode(n) \triangleq [s \in Slots \mapsto n.insts[s].voted]$$

$$PrepareReplyMsgs \triangleq [type : \{\text{“PrepareReply”}\}, src : Replicas, bal : Ballots, votes : InstsVotes]$$

$$PrepareReplyMsg(r, b, iv) \triangleq [type \mapsto \text{“PrepareReply”}, src \mapsto r, bal \mapsto b, votes \mapsto iv]$$

$$
\begin{array}{l}
PeakVotedWrite(prs, s) \triangleq \\
\quad \text{IF } \forall\, pr \in prs : pr.votes[s].bal = 0 \\
\qquad \text{THEN “nil”} \\
\qquad \text{ELSE LET } ppr \triangleq \\
\qquad\qquad\qquad \text{CHOOSE } ppr \in prs : \\
\qquad\qquad\qquad\quad \forall\, pr \in prs : pr.votes[s].bal \leq ppr.votes[s].bal \\
\qquad\qquad \text{IN } ppr.votes[s].write
\end{array}
$$

$$
\begin{array}{l}
LastTouchedSlot(prs) \triangleq \\
\quad \text{IF } \forall\, s \in Slots : PeakVotedWrite(prs, s) = \text{“nil”} \\
\qquad \text{THEN } 0 \\
\qquad \text{ELSE CHOOSE } s \in Slots : \\
\qquad\qquad \wedge\, PeakVotedWrite(prs, s) \neq \text{“nil”} \\
\qquad\qquad \wedge\, \forall\, t \in (s+1) \,..\, NumWrites : PeakVotedWrite(prs, t) = \text{“nil”}
\end{array}
$$

$$PrepareNoticeMsgs \triangleq [type : \{\text{“PrepareNotice”}\}, src : Replicas, bal : Ballots, commit\_prev : \{0\} \cup Slots]$$

$$PrepareNoticeMsg(r, b, cp) \triangleq [type \mapsto \text{“PrepareNotice”}, src \mapsto r,$$

$bal \mapsto b,$
$commit\_prev \mapsto cp]$

$AcceptMsgs \triangleq [type : \{\text{“Accept”}\}, src : Replicas,$
$bal : Ballots,$
$slot : Slots,$
$write : Writes]$

$AcceptMsg(r, b, s, c) \triangleq [type \mapsto \text{“Accept”}, src \mapsto r,$
$bal \mapsto b,$
$slot \mapsto s,$
$write \mapsto c]$

$AcceptReplyMsgs \triangleq [type : \{\text{“AcceptReply”}\}, src : Replicas,$
$bal : Ballots,$
$slot : Slots]$

$AcceptReplyMsg(r, b, s) \triangleq [type \mapsto \text{“AcceptReply”}, src \mapsto r,$
$bal \mapsto b,$
$slot \mapsto s]$

$CommitNoticeMsgs \triangleq [type : \{\text{“CommitNotice”}\}, upto : Slots]$

$CommitNoticeMsg(u) \triangleq [type \mapsto \text{“CommitNotice”}, upto \mapsto u]$

Lease protocol messages. Guard/$Renew$ carry the roster and a per-pair seq num; replies and $Revoke(Reply)$ carry only the ballot $+ seq$.

$GuardMsgs \triangleq [type : \{\text{“Guard”}\}, grantor : Replicas,$
$grantee : Replicas,$
$bal : Ballots,$
$ros : Rosters,$
$seq : SeqNums]$

$GuardMsg(f, p, b, ro, s) \triangleq [type \mapsto \text{“Guard”}, grantor \mapsto f,$
$grantee \mapsto p,$
$bal \mapsto b,$
$ros \mapsto ro,$
$seq \mapsto s]$

$GuardReplyMsgs \triangleq [type : \{\text{“GuardReply”}\}, grantee : Replicas,$
$grantor : Replicas,$
$bal : Ballots,$
$seq : SeqNums]$

$GuardReplyMsg(p, f, b, s) \triangleq [type \mapsto \text{“GuardReply”}, grantee \mapsto p,$
$grantor \mapsto f,$
$bal \mapsto b,$
$seq \mapsto s]$

$RenewMsgs \triangleq [type : \{\text{“Renew”}\}, grantor : Replicas,$
$grantee : Replicas,$
$bal : Ballots,$
$ros : Rosters,$
$seq : SeqNums]$

$RenewMsg(f, p, b, ro, s) \triangleq [type \mapsto \text{“Renew”}, grantor \mapsto f,$
$grantee \mapsto p,$
$bal \mapsto b,$

$ros \mapsto ro,$
$seq \mapsto s]$

$RenewReplyMsgs \triangleq [type : \{\text{“RenewReply”}\}, grantee : Replicas,$
$grantor : Replicas,$
$bal : Ballots,$
$seq : SeqNums]$

$RenewReplyMsg(p, f, b, s) \triangleq [type \mapsto \text{“RenewReply”}, grantee \mapsto p,$
$grantor \mapsto f,$
$bal \mapsto b,$
$seq \mapsto s]$

$RevokeMsgs \triangleq [type : \{\text{“Revoke”}\}, grantor : Replicas,$
$grantee : Replicas,$
$bal : Ballots,$
$seq : SeqNums]$

$RevokeMsg(f, p, b, s) \triangleq [type \mapsto \text{“Revoke”}, grantor \mapsto f,$
$grantee \mapsto p,$
$bal \mapsto b,$
$seq \mapsto s]$

$RevokeReplyMsgs \triangleq [type : \{\text{“RevokeReply”}\}, grantee : Replicas,$
$grantor : Replicas,$
$bal : Ballots,$
$seq : SeqNums]$

$RevokeReplyMsg(p, f, b, s) \triangleq [type \mapsto \text{“RevokeReply”}, grantee \mapsto p,$
$grantor \mapsto f,$
$bal \mapsto b,$
$seq \mapsto s]$

$$\begin{aligned} Messages \triangleq\ & PrepareMsgs \\ \cup\ & PrepareReplyMsgs \\ \cup\ & PrepareNoticeMsgs \\ \cup\ & AcceptMsgs \\ \cup\ & AcceptReplyMsgs \\ \cup\ & CommitNoticeMsgs \\ \cup\ & GuardMsgs \\ \cup\ & GuardReplyMsgs \\ \cup\ & RenewMsgs \\ \cup\ & RenewReplyMsgs \\ \cup\ & RevokeMsgs \\ \cup\ & RevokeReplyMsgs \end{aligned}$$

Main algorithm in *PlusCal*.

algorithm *Bodega*

variable $msgs = \{\},$ messages in the network
$node = [r \in Replicas \mapsto NullNode],$ replica merged state
$pending = InitPending,$ sequence of pending reqs
$observed = \langle\rangle,$ client observed events
$crashed = [r \in Replicas \mapsto \text{FALSE}],$ replica crashed flag
$time = [r \in Replicas \mapsto 1]$; per-node monotone clock

define

A lease from grantee p's perspective is "active" iff p's $asGrantee[f]$ is in *Renewed* status, un-expired, and recording roster ros.

$$FGrantsPWithRos(f, p, ros) \triangleq \begin{array}{l} \land node[p].asGrantee[f].status = \text{“Renewed”} \\ \land node[p].asGrantee[f].leaseExpire > time[p] \\ \land node[p].asGrantee[f].ros = ros \end{array}$$

True if a node r thinks of itself as leader of the latest roster.

$$\begin{array}{l} ThinkAmLeader(r) \triangleq \\ \quad \text{LET } ros \triangleq node[r].rosMaxKnown \\ \quad \text{IN } \begin{array}{l} \land node[r].leader = r \\ \land node[r].balPrepared = node[r].balMaxKnown \\ \land ros.bal = node[r].balMaxKnown \\ \land ros.leader = r \\ \land Cardinality(\{f \in Replicas : \\ \qquad FGrantsPWithRos(f, r, ros)\}) \\ \quad \geq MajorityNum \end{array} \end{array}$$

True if a node r thinks of itself as a follower of the latest roster.

$$\begin{array}{l} ThinkAmFollower(r) \triangleq \\ \quad \text{LET } ros \triangleq node[r].rosMaxKnown \\ \quad \text{IN } \begin{array}{l} \land node[r].leader \neq r \\ \land ros.bal = node[r].balMaxKnown \\ \land ros.leader \neq r \\ \land ros.leader \neq \text{“none”} \\ \land Cardinality(\{f \in Replicas : \\ \qquad FGrantsPWithRos(f, ros.leader, ros)\}) \\ \quad \geq MajorityNum \end{array} \end{array}$$

True if a node r thinks of itself as a responder of the latest roster.

$$\begin{array}{l} ThinkAmResponder(r) \triangleq \\ \quad \land ThinkAmFollower(r) \\ \quad \land r \in node[r].rosMaxKnown.responders \end{array}$$

The "safety threshold" condition: a node serves a read locally only if it has committed up to where previous leaders could have committed.

$$BallotTransferred(r) \triangleq node[r].commitUpTo \geq node[r].commitPrev$$

Given caller node r (a leader or a responder) and a set of $AcceptReplies$, decide whether the write is committable: majority quorum AND the caller's known roster's *responders* all voted.

$$\begin{array}{l} WriteCommittable(r, ars) \triangleq \\ \quad \text{LET } acceptors \triangleq \{ar.src : ar \in ars\} \\ \quad \text{IN } \begin{array}{l} \land Cardinality(acceptors) \geq MajorityNum \\ \land node[r].rosMaxKnown.responders \subseteq acceptors \end{array} \end{array}$$

When node r's ballot rises to nb, any live outgoing grant whose roster is at an older ballot must be revoked: *Guarding* grants reset locally, *Renewing* grants move to *Revoking* with *seq* bumped. Expired grants are untouched (cleared later by $TimeTick$).

$$\begin{array}{l} RevokeStaleAsGrantor(r, nb) \triangleq \\ \quad [p \in Replicas \mapsto \\ \qquad \text{LET } g \triangleq node[r].asGrantor[p] \\ \qquad \text{IN IF } \begin{array}{l} \land g.ros.bal < nb \\ \land g.status = \text{“Guarding”} \\ \land g.guardExpire > time[r] \end{array} \end{array}$$

```
                THEN [NullGrantorState EXCEPT !.seq = g.seq]
            ELSE IF ∧ g.ros.bal < nb
                    ∧ g.status = "Renewing"
                    ∧ g.leaseExpire > time[r]
                THEN [g EXCEPT !.status = "Revoking", !.seq = g.seq + 1]
            ELSE g]

    Revoke messages to emit alongside RevokeStaleAsGrantor(r, nb). Seq num
    matches the bumped seq in the Revoking state above.
    RevokeStaleSendMsgs(r, nb) ≜
        {RevokeMsg(r, p, nb, node[r].asGrantor[p].seq + 1) :
            p ∈ {p ∈ Replicas :
                    ∧ node[r].asGrantor[p].ros.bal < nb
                    ∧ node[r].asGrantor[p].status = "Renewing"
                    ∧ node[r].asGrantor[p].leaseExpire > time[r]}}

    Miscellaneous model checking helpers:
    reqsMade ≜ {e.cmd : e ∈ {e ∈ Range(observed) : e.type = "Req"}}

    acksRecv ≜ {e.cmd : e ∈ {e ∈ Range(observed) : e.type = "Ack"}}

    AppendObserved(seq) ≜
        LET filter(e) ≜ IF e.type = "Req" THEN e.cmd ∉ reqsMade
                                          ELSE e.cmd ∉ acksRecv
        IN observed ∘ SelectSeq(seq, filter)

    UnseenPending(r) ≜
        LET filter(c) ≜ ∀ s ∈ Slots : node[r].insts[s].write ≠ c
        IN SelectSeq(pending, filter)

    RemovePending(cmd) ≜
        LET filter(c) ≜ c ≠ cmd
        IN SelectSeq(pending, filter)

    terminated ≜ ∧ Len(pending) = 0
                 ∧ Cardinality(reqsMade) = NumCommands
                 ∧ Cardinality(acksRecv) = NumCommands

    numCrashed ≜ Cardinality({r ∈ Replicas : crashed[r]})

    timeExhausted ≜ ∀ r ∈ Replicas : time[r] = MaxTime
end define ;

Send a set of messages helper.
macro Send(set) begin
    msgs := msgs ∪ set ;
end macro ;

Observe client events helper.
macro Observe(seq) begin
    observed := AppendObserved(seq) ;
end macro ;

Resolve a pending command helper.
macro Resolve(c) begin
    pending := RemovePending(c) ;
end macro ;
```

```
Someone steps up as leader: picks a greater ballot, picks a fresh
responders set, and broadcasts Prepare. Also revokes any stale outgoing
lease grants: Guarding ones are reset locally; Renewing ones transition
to Revoking and send Revoke to the respective grantees.
macro BecomeLeader(r) begin
    if I'm not a current leader
    await node[r].leader ≠ r ;
    pick a greater ballot and an arbitrary responders set for new roster
    with b ∈ Ballots,
         resps ∈ SUBSET {f ∈ Replicas : f ≠ r},
         ros = Roster(b, r, resps)
     do
        await ∧ b > node[r].balMaxKnown
              ∧ ¬∃ m ∈ msgs : (m.type = "Prepare") ∧ (m.bal = b) ;
                    W.L.O.G., using this clause to model that ballot
                    numbers from different proposers be unique
        update states and restart Prepare phase for in-progress instances;
        also revoke any stale outgoing lease grants
        node[r].leader := r ||
        node[r].commitPrev := InfinitySlot ||
        node[r].balPrepared     := 0 ||
        node[r].balMaxKnown := b ||
        node[r].rosMaxKnown := ros ||
        node[r].insts :=
            [s ∈ Slots ↦
                [node[r].insts[s]
                    EXCEPT !.status = IF @ = "Accepting"
                                      THEN "Preparing"
                                      ELSE @]] ||
        node[r].asGrantor := RevokeStaleAsGrantor(r, b) ;
        broadcast Prepare and reply to myself instantly; also send Revokes
        for the just-Revoking grants
        Send(     {PrepareMsg(r, b),
                   PrepareReplyMsg(r, b, VotesByNode(node[r]))}
             ∪ RevokeStaleSendMsgs(r, b)) ;
    end with ;
end macro ;

Replica replies to a Prepare message. Also revokes any stale outgoing lease
grants the same way as BecomeLeader.
macro HandlePrepare(r) begin
    if receiving a Prepare message with larger ballot than ever seen
    with m ∈ msgs do
        await ∧ m.type = "Prepare"
              ∧ m.bal > node[r].balMaxKnown
              Prepare arrives along with an active lease from its sender;
              here we assert there's a Renewed grantee-side record for it.
              This stands in for "Prepare piggybacks the new roster"
              ∧ node[r].asGrantee[m.src].status = "Renewed"
              ∧ node[r].asGrantee[m.src].leaseExpire > time[r]
              ∧ node[r].asGrantee[m.src].ros.bal = m.bal ;
        with ros = node[r].asGrantee[m.src].ros do
            update states and reset statuses; also revoke stale outgoing
            lease grants
```

```
            node[r].leader := m.src ||
            node[r].commitPrev := InfinitySlot ||
            node[r].balMaxKnown := m.bal ||
            node[r].rosMaxKnown := ros ||
            node[r].insts :=
                [s ∈ Slots ↦
                    [node[r].insts[s]
                        EXCEPT !.status = IF @ = "Accepting"
                                          THEN "Preparing"
                                          ELSE @]] ||
            node[r].asGrantor := RevokeStaleAsGrantor(r, m.bal);
            send back PrepareReply with my voted list; also send Revokes
            for the just-Revoking grants
            Send(    {PrepareReplyMsg(r, m.bal, VotesByNode(node[r]))}
                  ∪ RevokeStaleSendMsgs(r, m.bal));
        end with ;
    end with ;
end macro ;

Leader gathers PrepareReply messages until condition met, then marks
the corresponding ballot as prepared and saves highest voted commands.
macro HandlePrepareReplies(r) begin
    if I'm waiting for PrepareReplies
    await ∧ node[r].leader = r
          ∧ node[r].balPrepared = 0;
    when there are enough number of PrepareReplies of desired ballot
    with prs = {m ∈ msgs : ∧ m.type = "PrepareReply"
                           ∧ m.bal = node[r].balMaxKnown}
    do
        await Cardinality({pr.src : pr ∈ prs}) ≥ MajorityNum;
        with prsGot ∈ {prsGot ∈ SUBSET prs :
                          Cardinality({pr.src : pr ∈ prsGot}) ≥ MajorityNum},
             lts = LastTouchedSlot(prsGot)
        do
            marks this ballot as prepared and saves highest voted command
            in each slot if any
            node[r].balPrepared := node[r].balMaxKnown ||
            node[r].insts :=
                [s ∈ Slots ↦
                    LET pvw ≜ PeakVotedWrite(prsGot, s)
                        adopted ≜ ∨ node[r].insts[s].status = "Preparing"
                                  ∨ ∧ node[r].insts[s].status = "Empty"
                                    ∧ pvw ≠ "nil"
                    IN  [node[r].insts[s]
                            EXCEPT !.status = IF adopted
                                              THEN "Accepting"
                                              ELSE @,
                                   !.write  = pvw,
                                   !.voted  = IF adopted
                                              THEN [bal ↦ node[r].balMaxKnown,
                                                    write ↦ pvw]
                                              ELSE @]] ||
            node[r].commitPrev := lts;
            send Accept messages for in-progress instances and reply to myself
```

```
            instantly; send PrepareNotices as well (mimicking threshold-carrying
            messages, in paper manuscript this is achieved via Guard messages)
            Send(   UNION
                    {{AcceptMsg(r, node[r].balPrepared, s, node[r].insts[s].write),
                      AcceptReplyMsg(r, node[r].balPrepared, s)} :
                     s ∈ {s ∈ Slots : node[r].insts[s].status = "Accepting"}}
                ∪ {PrepareNoticeMsg(r, node[r].balPrepared, lts)});
        end with ;
    end with ;
end macro ;

Follower receives PrepareNotice from a prepared and recovered leader,
updating its commitPrev accordingly.
macro HandlePrepareNotice(r) begin
    if I'm a follower waiting on PrepareNotice
    await ∧ ThinkAmFollower(r)
          ∧ node[r].commitPrev = InfinitySlot ;
    when there's a PrepareNotice message in effect
    with m ∈ msgs do
        await ∧ m.type = "PrepareNotice"
              ∧ m.bal = node[r].balMaxKnown ;
        update my commitPrev
        node[r].commitPrev := m.commit_prev ;
    end with ;
end macro ;

A prepared leader takes a new write request into the next empty slot.
macro TakeNewWriteRequest(r) begin
    if I'm a prepared leader and there's pending write request
    with unseen = UnseenPending(r) do
        await ∧ ThinkAmLeader(r)
              ∧ ∃ s ∈ Slots : node[r].insts[s].status = "Empty"
              ∧ Len(unseen) > 0
              ∧ Head(unseen) ∈ Writes ;
        find the next empty slot and pick a pending request
        with s = FirstEmptySlot(node[r].insts),
             c = Head(unseen)
         do
            update slot status and voted
            node[r].insts[s].status := "Accepting" ||
            node[r].insts[s].write := c ||
            node[r].insts[s].voted.bal := node[r].balPrepared ||
            node[r].insts[s].voted.write := c ;
            broadcast Accept and reply to myself instantly
            Send({AcceptMsg(r, node[r].balPrepared, s, c),
                  AcceptReplyMsg(r, node[r].balPrepared, s)}) ;
            append to observed events sequence if haven't yet
            Observe(⟨ReqEvent(c)⟩) ;
        end with ;
    end with ;
end macro ;

Replica replies to an Accept message. If the Accept's ballot is higher than
what we've known, also revokes any stale outgoing lease grants.
macro HandleAccept(r) begin
```

```
    if I'm a follower
    await ThinkAmFollower(r);
    if receiving an unreplied Accept message with valid ballot, piggybacked
    with an active lease from the sender
    with m ∈ msgs do
        await ∧ m.type = "Accept"
              ∧ m.bal ≥ node[r].balMaxKnown
              ∧ m.bal ≥ node[r].insts[m.slot].voted.bal
              ∧ node[r].asGrantee[m.src].status = "Renewed"
              ∧ node[r].asGrantee[m.src].leaseExpire > time[r]
              ∧ node[r].asGrantee[m.src].ros.bal = m.bal;
        update node states and corresponding instance's states; also
        revoke any stale outgoing lease grants
        node[r].leader := m.src ||
        node[r].balMaxKnown := m.bal ||
        node[r].rosMaxKnown := node[r].asGrantee[m.src].ros ||
        node[r].insts[m.slot].status := "Accepting" ||
        node[r].insts[m.slot].write := m.write ||
        node[r].insts[m.slot].voted.bal := m.bal ||
        node[r].insts[m.slot].voted.write := m.write ||
        node[r].asGrantor := RevokeStaleAsGrantor(r, m.bal);
        send back AcceptReply; also send Revokes for the just-Revoking grants
        Send(    {AcceptReplyMsg(r, m.bal, m.slot)}
             ∪  RevokeStaleSendMsgs(r, m.bal));
    end with ;
end macro ;

Leader gathers AcceptReply messages for a slot until condition met,
then marks the slot as committed and acknowledges the client.
macro HandleAcceptReplies(r) begin
    if I'm a prepared leader
    await ∧ ThinkAmLeader(r)
          ∧ node[r].commitUpTo < NumWrites
          ∧ node[r].insts[node[r].commitUpTo + 1].status = "Accepting";
              W.L.O.G., only enabling the next slot after commitUpTo
    for this slot, when there is a good set of AcceptReplies that is at
    least a majority number and that covers all responders
    with s = node[r].commitUpTo + 1,
         c = node[r].insts[s].write,
         ls = s − 1,
         v = IF ls = 0 THEN "nil" ELSE node[r].insts[ls].write,
         ars = {m ∈ msgs : ∧ m.type = "AcceptReply"
                           ∧ m.slot = s
                           ∧ m.bal = node[r].balPrepared}
     do
        await WriteCommittable(r, ars);
        with arsGot ∈ {arsGot ∈ SUBSET ars :
                          WriteCommittable(r, arsGot)}
         do
            marks this slot as committed and apply command
            node[r].insts[s].status := "Committed" ||
            node[r].commitUpTo := s;
            append to observed events sequence if haven't yet, and remove
            the command from pending
```

```
            Observe(⟨AckEvent(c, v, r)⟩);
            Resolve(c);
            broadcast CommitNotice to followers
            Send({CommitNoticeMsg(s)});
        end with ;
    end with ;
end macro ;

Replica receives new commit notification.
macro HandleCommitNotice(r) begin
    if I'm a follower waiting on CommitNotice
    await ∧ ThinkAmFollower(r)
          ∧ node[r].commitUpTo < NumWrites
          ∧ node[r].insts[node[r].commitUpTo + 1].status = "Accepting";
    for this slot, when there's a CommitNotice message
    with s = node[r].commitUpTo + 1,
         c = node[r].insts[s].write,
         m ∈ msgs
     do
        await ∧ m.type = "CommitNotice"
              ∧ m.upto = s;
        marks this slot as committed and apply command
        node[r].insts[s].status := "Committed" ||
        node[r].commitUpTo := s;
    end with ;
end macro ;

A prepared leader or a responder follower takes a new read request and
serves it locally.
macro TakeNewReadRequest(r) begin
    if I'm a caught-up leader or responder follower
    with unseen = UnseenPending(r) do
        await ∧ ∨ ThinkAmLeader(r)
                ∨ ThinkAmResponder(r)
              ∧ BallotTransferred(r)
              ∧ Len(unseen) > 0
              ∧ Head(unseen) ∈ Reads;
        pick a pending request; examine my log and find the last non-empty
        slot, check its status
        with s = LastNonEmptySlot(node[r].insts),
             v = IF s = 0 THEN "nil" ELSE node[r].insts[s].write,
             c = Head(unseen)
         do
            if the latest value is in Committed status, can directly reply;
            otherwise, should hold until I've received enough broadcasted
            AcceptReplies indicating that the write is surely to be committed
            await ∨ s = 0
                  ∨ s > 0 ∧ node[r].insts[s].status = "Committed"
                  ∨ LET ars ≜ {m ∈ msgs : ∧ m.type = "AcceptReply"
                                          ∧ m.slot = s
                                          ∧ m.bal = node[r].balMaxKnown}
                    IN WriteCommittable(r, ars);
            acknowledge client with the latest value, and remove the command
            from pending
            Observe(⟨ReqEvent(c), AckEvent(c, v, r)⟩);
```

```
            Resolve(c);
        end with ;
    end with ;
end macro ;

Replica node crashes itself under promised conditions.
macro ReplicaCrashes(r) begin
    if less than (N − MajorityNum) number of replicas have failed
    await ∧ MajorityNum + numCrashed < Cardinality(Replicas)
          ∧ ¬crashed[r]
          ∧ node[r].balMaxKnown < MaxBallot;
    mark myself as crashed
    crashed[r] := TRUE;
end macro ;

Grantor f opens new leases to peers for its known current roster. Gated
by the condition that f has a non-null rosMaxKnown and f is not actively
granting to anyone.
macro GrantorInitiateLeases(f) begin
    await ∧ node[f].rosMaxKnown ≠ NullRoster
          ∧ ∀ p ∈ Replicas : node[f].asGrantor[p].status = "None";
    node[f].asGrantor :=
        [p ∈ Replicas ↦
            [status ↦ "Guarding",
             guardExpire ↦ time[f] + TGuard,
             leaseExpire ↦ 0,
             ros ↦ node[f].rosMaxKnown,
             seq ↦ node[f].asGrantor[p].seq + 1]];
    Send({GuardMsg(f, p,
                   node[f].balMaxKnown,
                   node[f].rosMaxKnown,
                   node[f].asGrantor[p].seq + 1) :
          p ∈ Replicas});
end macro ;

Grantee p receives a Guard from grantor f. Accept iff ballot is at least as high as
p's view and the message's seq is strictly higher than any seq ever observed on
this pair. If ballot rises, also revokes stale outgoing lease grants from p.
macro HandleGuard(p) begin
    with m ∈ msgs do
        await ∧ m.type = "Guard"
              ∧ m.grantee = p
              ∧ m.bal ≥ node[p].balMaxKnown
              ∧ m.seq > node[p].asGrantee[m.grantor].seq
              ∧ node[p].asGrantee[m.grantor].status = "None";
        update ballot in case higher, and start Guarded state; bump seq;
        also revoke any stale outgoing lease grants
        node[p].balMaxKnown := m.bal ||
        node[p].asGrantee[m.grantor] :=
            [status ↦ "Guarded",
             guardExpire ↦ time[p] + TGuard,
             leaseExpire ↦ 0,
             ros ↦ m.ros,
             seq ↦ m.seq + 1] ||
        node[p].asGrantor := RevokeStaleAsGrantor(p, m.bal);
```

```
        reply GuardReply back to grantor, stamped with the new seq; also
        send Revokes for the just-Revoking outgoing grants
        Send(      {GuardReplyMsg(p, m.grantor, m.bal, m.seq + 1)}
              ∪  RevokeStaleSendMsgs(p, m.bal));
    end with ;
end macro ;

Grantor f receives a GuardReply: transition asGrantor[m.grantee] from
Guarding to Renewing; send first Renew.
macro HandleGuardReply(f) begin
    with m ∈ msgs do
        await ∧ m.type = "GuardReply"
              ∧ m.grantor = f
              ∧ m.bal = node[f].balMaxKnown
              ∧ m.seq > node[f].asGrantor[m.grantee].seq
              ∧ node[f].asGrantor[m.grantee].status = "Guarding";
        loose expiry for safety, tightened at first RenewReply
        node[f].asGrantor[m.grantee] :=
            [node[f].asGrantor[m.grantee] EXCEPT
                ! .status = "Renewing",
                ! .leaseExpire = time[f] + TGuard + TLease,
                ! .seq = m.seq + 1];
        Send({RenewMsg(f, m.grantee, m.bal,
                       node[f].rosMaxKnown, m.seq + 1)});
    end with ;
end macro ;

Grantor f spontaneously sends subsequent Renews to its current grantees.
macro SpontaneousRenew(f) begin
    await ∃ p ∈ Replicas :
            ∧ node[f].asGrantor[p].ros.bal = node[f].balMaxKnown
            ∧ node[f].asGrantor[p].status = "Renewing"
            ∧ time[f] < node[f].asGrantor[p].leaseExpire
            ∧ time[f] + TLease > node[f].asGrantor[p].leaseExpire;
    with ps = {p ∈ Replicas :
                  ∧ node[f].asGrantor[p].ros.bal = node[f].balMaxKnown
                  ∧ node[f].asGrantor[p].status = "Renewing"
                  ∧ time[f] < node[f].asGrantor[p].leaseExpire
                  ∧ time[f] + TLease > node[f].asGrantor[p].leaseExpire} do
        node[f].asGrantor :=
            [p ∈ Replicas ↦
                IF p ∈ ps
                  THEN [node[f].asGrantor[p] EXCEPT
                          !.leaseExpire = node[f].asGrantor[p].leaseExpire + TLease,
                          !.seq = node[f].asGrantor[p].seq + 1]
                  ELSE node[f].asGrantor[p]];
        Send({RenewMsg(f, p,
                       node[f].balMaxKnown,
                       node[f].rosMaxKnown,
                       node[f].asGrantor[p].seq + 1) :
              p ∈ ps});
    end with ;
end macro ;

Grantee p receives a Renew.
```

```
macro HandleRenew(p) begin
    with m ∈ msgs do
        await ∧ m.type = "Renew"
              ∧ m.grantee = p
              ∧ m.bal = node[p].balMaxKnown
              ∧ m.seq > node[p].asGrantee[m.grantor].seq
              ∧ ∨ ∧ node[p].asGrantee[m.grantor].status = "Guarded"
                  ∧ time[p] < node[p].asGrantee[m.grantor].guardExpire
                ∨ ∧ node[p].asGrantee[m.grantor].status = "Renewed"
                  ∧ time[p] < node[p].asGrantee[m.grantor].leaseExpire
                  ∧ time[p] + TLease > node[p].asGrantee[m.grantor].leaseExpire ;
        node[p].asGrantee[m.grantor] :=
            [node[p].asGrantee[m.grantor] EXCEPT
                ! .status = "Renewed",
                ! .leaseExpire = time[p] + TLease,
                ! .ros = m.ros,
                ! .seq = m.seq + 1] ;
        Send({RenewReplyMsg(p, m.grantor, m.bal, m.seq + 1)}) ;
    end with ;
end macro ;

Grantor f receives a RenewReply; tightens down the loose expiry.
macro HandleRenewReply(f) begin
    with m ∈ msgs do
        await ∧ m.type = "RenewReply"
              ∧ m.grantor = f
              ∧ m.bal = node[f].balMaxKnown
              ∧ m.seq > node[f].asGrantor[m.grantee].seq
              ∧ node[f].asGrantor[m.grantee].status = "Renewing"
              ∧ time[f] + TLease > node[f].asGrantor[m.grantee].leaseExpire ;
        node[f].asGrantor[m.grantee] :=
            [node[f].asGrantor[m.grantee] EXCEPT
                ! .leaseExpire = time[f] + TLease,
                ! .seq = m.seq] ;
    end with ;
end macro ;

Grantee p processes a Revoke from grantor f. If ballot rises, also
revokes stale outgoing lease grants from p.
macro HandleRevoke(p) begin
    with m ∈ msgs do
        await ∧ m.type = "Revoke"
              ∧ m.grantee = p
              ∧ m.bal ≥ node[p].balMaxKnown
              ∧ m.seq > node[p].asGrantee[m.grantor].seq
              ∧ node[p].asGrantee[m.grantor].status ∈ {"Guarded", "Renewed"} ;
        update ballot in case higher, clear grantee state but preserve seq;
        also revoke any stale outgoing lease grants
        node[p].balMaxKnown := m.bal ||
        node[p].asGrantee[m.grantor] :=
            [NullGranteeState EXCEPT !.seq = m.seq + 1] ||
        node[p].asGrantor := RevokeStaleAsGrantor(p, m.bal) ;
        Send(     {RevokeReplyMsg(p, m.grantor, m.bal, m.seq + 1)}
              ∪ RevokeStaleSendMsgs(p, m.bal)) ;
    end with ;
```

```
end macro ;

Grantor f receives a RevokeReply; drops the lease promptly.
macro HandleRevokeReply(f) begin
    with m ∈ msgs do
        await ∧ m.type = "RevokeReply"
              ∧ m.grantor = f
              ∧ m.bal = node[f].balMaxKnown
              ∧ m.seq > node[f].asGrantor[m.grantee].seq
              ∧ node[f].asGrantor[m.grantee].status = "Revoking" ;
        node[f].asGrantor[m.grantee] :=
            [NullGrantorState EXCEPT !.seq = m.seq] ;
    end with ;
end macro ;

Advances time by one tick globally, and garbage-collects expired lease
state. Per-pair seq counters are preserved across GC.
macro TimeTick() begin
    await ∀ r ∈ Replicas : time[r] < MaxTime ;
    time := [r ∈ Replicas ↦ time[r] + 1] ;
    node := [r ∈ Replicas ↦
        [node[r] EXCEPT
            ! .asGrantor =
                [p ∈ Replicas ↦
                    IF ∨ ∧ node[r].asGrantor[p].status ∈ {"Renewing", "Revoking"}
                         ∧ node[r].asGrantor[p].leaseExpire ≤ time[r]
                       ∨ ∧ node[r].asGrantor[p].status = "Guarding"
                         ∧ node[r].asGrantor[p].guardExpire ≤ time[r]
                    THEN [NullGrantorState EXCEPT !.seq = node[r].asGrantor[p].seq]
                    ELSE node[r].asGrantor[p]],
            ! .asGrantee =
                [f ∈ Replicas ↦
                    IF ∨ ∧ node[r].asGrantee[f].status = "Renewed"
                         ∧ node[r].asGrantee[f].leaseExpire ≤ time[r]
                       ∨ ∧ node[r].asGrantee[f].status = "Guarded"
                         ∧ node[r].asGrantee[f].guardExpire ≤ time[r]
                    THEN [NullGranteeState EXCEPT !.seq = node[r].asGrantee[f].seq]
                    ELSE node[r].asGrantee[f]]]] ;
end macro ;

Replica server node main loop: consensus actions + lease protocol actions
+ global time tick are all available via either/or.
process Replica ∈ Replicas
 begin
    rloop: while (¬terminated) ∧ (¬timeExhausted) ∧ (¬crashed[self]) do
        either
            BecomeLeader(self) ;
        or
            HandlePrepare(self) ;
        or
            HandlePrepareReplies(self) ;
        or
            HandlePrepareNotice(self) ;
        or
            TakeNewWriteRequest(self) ;
```

```
        or
            HandleAccept(self);
        or
            HandleAcceptReplies(self);
        or
            HandleCommitNotice(self);
        or
            TakeNewReadRequest(self);
        or
            GrantorInitiateLeases(self);
        or
            HandleGuard(self);
        or
            HandleGuardReply(self);
        or
            SpontaneousRenew(self);
        or
            HandleRenew(self);
        or
            HandleRenewReply(self);
        or
            HandleRevoke(self);
        or
            HandleRevokeReply(self);
        or
            TimeTick();
        or
            if NodeFailuresOn then
                ReplicaCrashes(self);
            end if ;
        end either ;
    end while ;
end process ;

end algorithm ;
```

```
MODULE Bodega_MC
EXTENDS Bodega

TLC config-related defs.
ConditionalPerm(set) ≜ IF Cardinality(set) > 1
                       THEN Permutations(set)
                       ELSE {}

SymmetricPerms ≜ ConditionalPerm(Replicas)
               ∪ ConditionalPerm(Writes)
               ∪ ConditionalPerm(Reads)

ConstMaxBallot ≜ 2

ConstTGuard ≜ 1
ConstTLease ≜ 1
ConstMaxTime ≜ 3
```

Type check invariant.

$$
\begin{aligned}
TypeOK \triangleq\ & \land \forall\, m \in msgs : m \in Messages \\
& \land \forall\, r \in Replicas : node[r] \in NodeStates \\
& \land \forall\, r \in Replicas : time[r] \in Times \\
& \land Len(pending) \leq NumCommands \\
& \land Cardinality(Range(pending)) = Len(pending) \\
& \land \forall\, c \in Range(pending) : c \in Commands \\
& \land Len(observed) \leq 2 * NumCommands \\
& \land Cardinality(Range(observed)) = Len(observed) \\
& \land Cardinality(reqsMade) \geq Cardinality(acksRecv) \\
& \land \forall\, e \in Range(observed) : e \in ClientEvents \\
& \land \forall\, r \in Replicas : crashed[r] \in \text{BOOLEAN}
\end{aligned}
$$

THEOREM $Spec \Rightarrow \Box\, TypeOK$

Lease expiration safety property.

$$
\begin{aligned}
& LeaseExpirationSafety \triangleq \\
& \quad \forall\, f, p \in Replicas : \\
& \qquad (\land node[p].asGrantee[f].status = \text{“Renewed”} \\
& \qquad\ \land node[p].asGrantee[f].leaseExpire > time[p]) \\
& \qquad\quad \Rightarrow (\land node[f].asGrantor[p].status \in \{\text{“Renewing”}, \text{“Revoking”}\} \\
& \qquad\qquad \land node[f].asGrantor[p].leaseExpire \\
& \qquad\qquad\quad \geq node[p].asGrantee[f].leaseExpire)
\end{aligned}
$$

THEOREM $Spec \Rightarrow \Box\, LeaseExpirationSafety$

Lease uniqueness guarantee assertions.

$$
\begin{aligned}
& AtMostGrantsOneRoster \triangleq \\
& \quad \forall\, f \in Replicas, b \in Ballots : \\
& \qquad Cardinality(\{ros \in Rosters : \\
& \qquad\qquad \exists\, p \in Replicas : \land FGrantsPWithRos(f, p, ros) \\
& \qquad\qquad\qquad\qquad \land ros.bal = b\}) \leq 1
\end{aligned}
$$

$$
\begin{aligned}
& AtMostOneStableRoster \triangleq \\
& \quad \forall\, ros1, ros2 \in Rosters : \\
& \qquad (\land Cardinality(\{f \in Replicas : \\
& \qquad\qquad\qquad \exists\, p \in Replicas : FGrantsPWithRos(f, p, ros1)\}) \\
& \qquad\qquad \geq MajorityNum \\
& \qquad\ \land Cardinality(\{f \in Replicas : \\
& \qquad\qquad\qquad \exists\, p \in Replicas : FGrantsPWithRos(f, p, ros2)\}) \\
& \qquad\qquad \geq MajorityNum) \\
& \qquad \Rightarrow (ros1 = ros2)
\end{aligned}
$$

$$
\begin{aligned}
\text{THEOREM } Spec \Rightarrow\ & \land \Box\, AtMostGrantsOneRoster \\
& \land \Box\, AtMostOneStableRoster
\end{aligned}
$$

Linearizability constraint.

$$
\begin{aligned}
ReqPosOfCmd(c) \triangleq\ & \text{CHOOSE } i \in 1 \,..\, Len(observed) : \\
& \land observed[i].type = \text{“Req”} \\
& \land observed[i].cmd = c
\end{aligned}
$$

$$AckPosOfCmd(c) \triangleq \text{CHOOSE } i \in 1 \,..\, Len(observed) :$$
$$\wedge\, observed[i].type = \text{“Ack”}$$
$$\wedge\, observed[i].cmd = c$$

$$ResultOfCmd(c) \triangleq observed[AckPosOfCmd(c)].val$$

$$OrderIdxOfCmd(order, c) \triangleq \text{CHOOSE } j \in 1 \,..\, Len(order) : order[j] = c$$

$$LastWriteBefore(order, j) \triangleq$$
$$\text{LET } k \triangleq \text{CHOOSE } k \in 0 \,..\, (j-1) :$$
$$\wedge\, (k = 0 \vee order[k] \in Writes)$$
$$\wedge\, \forall\, l \in (k+1) \,..\, (j-1) : order[l] \in Reads$$
$$\text{IN IF } k = 0 \text{ THEN “nil” ELSE } order[k]$$

$$IsLinearOrder(order) \triangleq$$
$$\wedge\, \{order[j] : j \in 1 \,..\, Len(order)\} = Commands$$
$$\wedge\, \forall\, j \in 1 \,..\, Len(order) :$$
$$ResultOfCmd(order[j]) = LastWriteBefore(order, j)$$

$$ObeysRealTime(order) \triangleq$$
$$\forall\, c1, c2 \in Commands :$$
$$(AckPosOfCmd(c1) < ReqPosOfCmd(c2))$$
$$\Rightarrow (OrderIdxOfCmd(order, c1) < OrderIdxOfCmd(order, c2))$$

$$Linearizability \triangleq$$
$$terminated \Rightarrow$$
$$\exists\, order \in [1 \,..\, NumCommands \to Commands] :$$
$$\wedge\, IsLinearOrder(order)$$
$$\wedge\, ObeysRealTime(order)$$

$$\text{THEOREM } Spec \Rightarrow Linearizability$$

Bodega_MC.cfg

```
SPECIFICATION Spec

CONSTANTS
    Replicas = {s1, s2, s3}
    Writes = {w1, w2}
    Reads = {r1, r2}
    MaxBallot <- ConstMaxBallot
    NodeFailuresOn <- TRUE
    TGuard <- ConstTGuard
    TLease <- ConstTLease
    MaxTime <- ConstMaxTime

SYMMETRY SymmetricPerms

INVARIANTS
    TypeOK
    LeaseExpirationSafety
    AtMostGrantsOneRoster
    AtMostOneStableRoster
    Linearizability

CHECK_DEADLOCK FALSE
```