LeaseGuard: Raft Leases Done Right

A. JESSE JIRYU DAVIS, MongoDB Research, USA
MURAT DEMIRBAS, MongoDB Research, USA
LINGZHI DENG, MongoDB, Inc., USA

Raft is a leading consensus algorithm for replicating writes in distributed databases. However, distributed databases also require consistent *reads*. To guarantee read consistency, a Raft-based system must either accept the high communication overhead of a safety check for each read, or implement *leader leases*. Prior lease protocols are vaguely specified and hurt availability, so most Raft systems implement them incorrectly or not at all. We introduce LeaseGuard, a novel lease algorithm that relies on guarantees specific to Raft elections. LeaseGuard is simple, rigorously specified in TLA+, and includes two novel optimizations that maximize availability during leader failover. The first optimization restores write throughput quickly, and the second improves read availability. We evaluate LeaseGuard with a simulation in Python and an implementation in LogCabin, the C++ reference implementation of Raft. By replacing LogCabin's default consistency mechanism (quorum checks), LeaseGuard reduces the overhead of consistent reads from one to zero network roundtrips. It also improves write throughput from ~1000 to ~10,000 writes per second, by eliminating contention between writes and quorum reads. Whereas traditional leases ban all reads on a new leader while it waits for a lease, in our LeaseGuard test the new leader instantly allows 99% of reads to succeed.

CCS Concepts: • **Computer systems organization** → **Cloud computing**; **Availability**; • **Theory of computation** → **Distributed algorithms**.

Additional Key Words and Phrases: Lease, Raft, Clock, Consensus

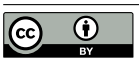
ACM Reference Format:
A. Jesse Jiryu Davis, Murat Demirbas, and Lingzhi Deng. 2026. LeaseGuard: Raft Leases Done Right. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 49 (February 2026), 25 pages. https://doi.org/10.1145/3786663

1 Introduction

The Raft consensus algorithm is used in MongoDB [57], CockroachDB [49], TiDB [25], YugabyteDB [51], and others [4, 16, 29]. It reliably replicates a log of write commands and handles leader failures. But replication introduces the risk of reading from a stale replica. If a database served stale reads while promising strong consistency, it would break its promise. For example, CockroachDB and YugabyteDB promise serializability and strongly consistent reads by default; TiDB and MongoDB support snapshot isolation. Violating these guarantees would lead to application misbehavior [15], lost updates [20], and security vulnerabilities [55], so the database needs a mechanism to ensure it reads from a fresh replica.

It is surprisingly hard to ensure consistent reads from a Raft leader. The leader is elected at the start of a *term* and executes all reads and writes until it crashes or is *deposed*, i.e., a new leader is elected with a higher term number. The deposed leader may be unaware for some time that it has been deposed, for example if it is on the minority side of a network partition. If it continues serving reads while the new leader executes writes, the deposed leader returns stale data, violating

Authors' Contact Information: A. Jesse Jiryu Davis, jesse@mongodb.com, MongoDB Research, New York City, NY, USA; Murat Demirbas, murat.demirbas@mongodb.com, MongoDB Research, New York City, NY, USA; Lingzhi Deng, lingzhi.deng@mongodb.com, MongoDB, Inc., New York City, NY, USA.



ACM 2836-6573/2026/2-ART49
https://doi.org/10.1145/3786663

consistency. Although elections may be rare in a single Raft group, large deployments make "rare" events common. MongoDB's public DBaaS fleet has about 200,000 Raft replica sets, of which half experience an election each week. Half of those elections are related to routine maintenance, and half are unplanned. A system of this size requires some mechanism to avoid stale reads during elections. Unfortunately, Raft's original mechanism [42] is expensive: before responding to each read request, a leader exchanges messages with a majority of the replica set to confirm it is still the highest-term leader. This quorum check incurs latency, I/O contention, and monetary costs in the cloud. It is a high price to pay for every query, just to ensure consistency during occasional elections.

Leader leases have been proposed [41] to make Raft reads consistent and efficient. A lease ensures that at most one node believes it is the leader at a time, so it can serve consistent reads locally without talking to other nodes. Unfortunately, the original description of Raft leases is vague, without code or formal specifications (see Section 8). The same applies to leader leases in Paxos, Raft's predecessor [10, 14]. For this reason, Raft systems often implement leases with bugs. HashiCorp's Raft implementation, for example, does not follow the Raft paper. It can violate consistency due to message delays, or because a new leader does not know of a prior leader's lease [31, 52]. These bugs were reported in 2016. The etcd Raft implementation also allows stale reads due to miscalculations of lease timeouts [5]. This bug was reported in 2024. All these bugs were still open at the time of writing. Most Raft implementations forego leases. They use expensive quorum checks, or they do not guarantee consistency at all. MongoDB's Raft variant [57] avoids leases: if a user demands consistent reads, the leader writes an empty entry to its log for *each* read request, and waits for majority replication of the entry to confirm it is not deposed.

Prior lease designs also hurt availability: a deposed leader's lease must expire before a new leader can process reads and writes. (E.g., TiDB's Raft lease causes 10 seconds of unavailability after an election [34].) During this period, blocked operations may queue up, then overwhelm the leader as soon as it acquires a lease; and this thundering herd could in turn cause a metastable failure [26]. Prior designs also risk gray failures, where a leader may continue renewing its lease, but fail to execute writes due to internal issues such as disk failure. The system is stuck with a "faux leader" that cannot make progress yet refuses to step aside.

LeaseGuard is a new lease protocol for Raft which solves the problems above. It is specified in detail, and it is simpler than previous lease designs: there are no changes to Raft messages, no additional messages, no new data structures, and no changes to the election protocol. Instead, in LeaseGuard **the log is the lease**. The leader establishes its lease by committing a log entry, and followers learn of the lease by replicating the log normally. We can rely on the log due to a Raft-specific guarantee called **Leader Completeness**: a newly elected leader is guaranteed to have all log entries that were replicated by a majority in the previous term. (Competing consensus protocols like Paxos and Viewstamped Replication [36] do not guarantee Leader Completeness.) This guarantee enables a simpler lease design, reduces implementation bugs, and enables availability optimizations overlooked in prior work. It also solves the gray-failure/faux-leader problem: only a leader who can make real progress (by writing and replicating log entries) can maintain its lease.

It is critical for distributed databases to maintain high availability and reliable performance, and avoid metastable failures. LeaseGuard minimizes write unavailability through its **deferred-commit writes** optimization (Section 3.2), allowing the new leader to write and replicate log entries while it waits for the deposed leader's lease to expire. LeaseGuard also minimizes read unavailability through its **inherited lease reads** optimization, enabling a new leader to serve some consistent reads concurrently with a deposed leader during the deposed leader's lease period, with zero communication overhead. This optimization in particular requires bounded-uncertainty clocks. Such clocks have recently become available in the cloud. Our formal TLA+ [33] specification and

correctness proofs (Section 4) validate the safety of these optimizations. Notably, we discovered these surprising optimizations through our exploration of leases in TLA+. Our improvements to read availability and write throughput together decrease the risk that a thundering herd of operations will overwhelm a new leader as soon as it acquires a lease.

To evaluate performance, we developed a Python simulation[1] of Raft and LeaseGuard (Section 6). The simulator models nodes as Python objects that communicate asynchronously, similar to Python's `asyncio` library. It enables deterministic, reproducible experiments given a set of parameters and a random seed. We also implemented a linearizability checker to validate protocol behavior.

For real-world validation, we implemented LeaseGuard in C++[2] by modifying LogCabin, a key-value database atop the reference implementation of Raft (Section 7). LogCabin lacked any lease mechanism until now. Our experiments evaluate LeaseGuard's impact on read and write latency, its availability after leader crashes, and how workload skewness affects read availability post-election. The results demonstrate that LeaseGuard achieves microsecond-latency local linearizable reads compared to many milliseconds of latency with LogCabin's default consistency mechanism, quorum checks. By avoiding quorum checks, LeaseGuard reduces the overhead of consistent reads from one to zero network roundtrips. It also improves write throughput from ~1000 to ~10,000 writes per second, by eliminating contention between writes and quorum reads. LeaseGuard improves read and write availability compared to traditional leases. Whereas traditional leases ban all reads on a new leader while it waits for a lease, in our LeaseGuard test the new leader instantly allows 99% of reads to succeed.

2 Background

2.1 Raft

In Raft, each node has a *state*: "leader", "candidate", or "follower". Each node has a *term* that tracks the highest term number it has seen. Nodes gossip their term numbers during every communication. To run for election, a follower increments its term and becomes a candidate, then requests votes from a majority of the replica set. Once elected, a node remains a leader until it crashes or observes another node with a higher term.

Clients invoke commands on the leader, which records them as *entries* in its *log* and sends them to followers in AppendEntries messages. Followers record entries in their own logs in the same order. An entry's *log index* is its position in the log. Each node has a *commitIndex*, the index of the latest entry it knows is durable. When the leader learns that a majority of nodes (including itself) have replicated up to a given log index, it advances its commitIndex to that point, i.e. it *commits* the entry and all previous entries. It applies committed entries' commands to its local state machine and updates its *lastApplied* index. The leader then replies to waiting clients, confirming their commands have succeeded. Followers eventually learn the leader's new commitIndex, which the leader sends them in subsequent AppendEntries messages. Thus followers' commitIndexes are less than or equal to the leader's in normal operation (Figure 1).

Raft and its predecessor MultiPaxos are similar [24, 54], but their election protocols differ significantly. In MultiPaxos, any node can be elected, and it must transfer from other nodes any log entries it lacks before it is fully functional. Raft, however, guarantees that a node is elected only if its log already includes all committed entries ("Leader Completeness"). Our work depends on this guarantee to provide novel simplifications and optimizations for leases.

[1] https://github.com/muratdem/RaftLeaderLeases/tree/main/Python
[2] https://github.com/mongodb-labs/logcabin/tree/leaseguard

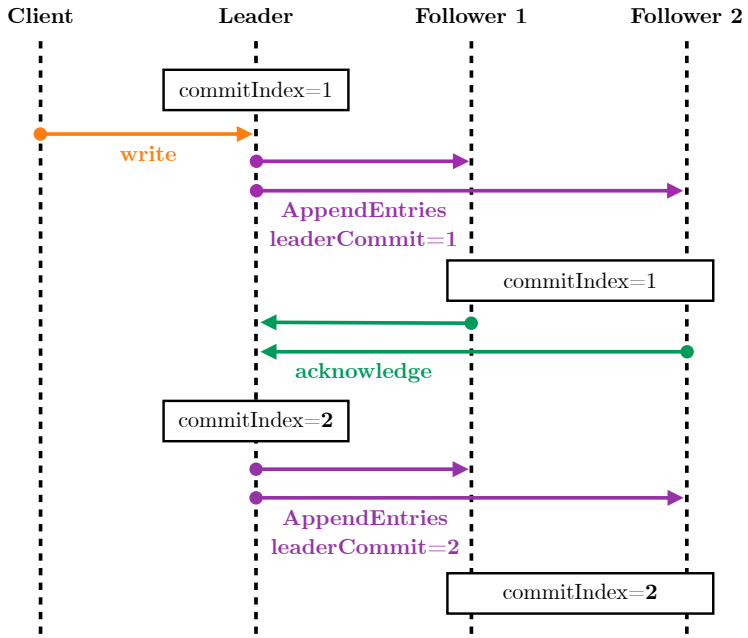


Fig. 1. Raft replication. The leader advances its commitIndex when it learns that an entry is majority-replicated. Followers learn the new commitIndex from later AppendEntries messages.

State Machine Replication protocols provide strongly-consistent, fault-tolerant write replication. But it is challenging to guarantee that reads conform to the strongest consistency level, linearizability [22]. Linearizability requires that (1) each operation appears to execute at a single instant between its invocation (when the client submits it) and completion (when the client receives the response), and (2) the execution order at these instants forms a valid sequential history, as if operations were executed atomically by a single thread. As discussed in the introduction, a deposed leader in Raft may violate linearizability, if it serves a stale read assuming it is still the leader.

2.2 Clocks with bounded uncertainty

We assume each node has access to some function *intervalNow()* which returns the interval [*earliest*, *latest*]. It is guaranteed that the true time was in this interval for at least a moment between the function's invocation and completion. Our lease algorithm requires a node to decide if a time recorded on another node is now more than Δ old, for some duration Δ. For any two time intervals t_1 and t_2, a node knows that t_1 is more than Δ old if *intervalNow()* has returned t_2 and $t_1.latest + \Delta < t_2.earliest$.

Google's Spanner paper [14] popularized the concept of bounded-uncertainty clocks in the cloud. It describes the TrueTime API, which returns an interval that is guaranteed to increase monotonically and to include the true time. Each Google data center has multiple time master machines; most have GPS receivers and the remainder have atomic clocks. Each Spanner server keeps its clock tightly synchronized, with known error bounds, by communicating with multiple time master machines and applying a variant of Marzullo's algorithm [37]. TrueTime's average error was 4ms in 2012. Presumably its precision has improved, but it is still available only for Google's own systems, not Google Cloud Platform customers. Meta has deployed a similar infrastructure, also for their exclusive use, with sub-microsecond error [40].

Starting in 2023, Amazon has deployed clocks with microsecond precision for public use. AWS TimeSync combines GPS receivers, the Precision Time Protocol [1], and a physical hardware clock on each server to provide a high-precision time signal [2]. Amazon's open source clock-bound [27] daemon monitors the local clock and calculates reliable error bounds. The AWS servers used in our evaluation have an average clock error less than 50µs.

The Huygens protocol [17] and its commercialization, Clockwork [28], provide a similar guarantee, and they can be deployed on any cloud. As more cloud providers recognize the value of precise, bounded-uncertainty clocks, we hope and expect they will become more widely available.

3 Adding leases to Raft

This section describes LeaseGuard using prose, pseudocode, and diagrams. A TLA+ model of LeaseGuard is available on GitHub[3]. We defer the correctness arguments to Section 4.

Configuration: All nodes have the same lease duration Δ.

Writes: A leader can always accept a write command. It creates a log entry that includes the command, and replicates it to followers, as in Raft. In LeaseGuard, the leader includes its current time interval, `intervalNow()`, in the entry (Figure 2 lines 5-6).

Advancing the commitIndex: As in Raft, a leader can advance its commitIndex to i once a majority of nodes have replicated the entry at i. Once an entry is committed and applied, the leader acknowledges it to the client (lines 13 and 40-42). LeaseGuard adds a restriction: the leader L_1 of term t cannot advance its commitIndex if L_1 has an entry $< \Delta$ old with term $< t$ from a deposed leader L_0 (lines 34-38). This prevents L_1 from committing writes while L_0 may have an active lease and may be executing reads, see Figure 4.

Reads: A leader L can serve a local linearizable read if it has a committed entry $< \Delta$ old in any term, because it knows no newer leader is committing writes (lines 18-21). If L has not yet committed an entry in its term, there is a "limbo region" of entries past L's commitIndex, up to and including the last entry L had when it was elected. L does not yet know if an earlier leader committed any of these entries, so L executes a read only if its result is not affected by any entry in the limbo region (lines 23-25).

Followers: LeaseGuard does not change follower behavior compared to vanilla Raft. Only when a follower is elected as a leader does it make use of the lease information implied by its log.

Elections: In contrast to previous work [14, 36, 44, 49], we leave Raft's election protocol [42] unmodified. Since our priority is to maximize availability, we allow a new leader to be elected before the old leader's lease expires. Even a node that knows of a valid lease may vote, become a candidate, and/or become a leader.

Figure 2 presents the pseudocode for LeaseGuard. The following sections focus on certain details of LeaseGuard.

3.1 Establishing a lease

Leases are naturally established and extended by Raft's replication protocol. A leader L_1 establishes a lease on followers by creating an entry e in its log and sending it to followers. When L_1 commits e (after hearing acks from a majority of nodes), L_1 can serve local linearizable reads while e is at most Δ old, because it knows no future leader will advance the commitIndex until e is more than Δ old. It knows this because, once e is committed, Leader Completeness implies that any leader L_2 in a future term will have e in its log, and thus know of L_1's lease. Later entries after e, including ordinary client write commands, automatically extend the lease.

[3]https://github.com/muratdem/RaftLeaderLeases/tree/main/TLA

```
# Handle a write request from a client.
def ClientWrite(command):
  if self.state != "leader": return "not leader"
  # Create new entry, log it and record its index.
  entry = (self.term, command, intervalNow())
  index = self.log.append(entry)
  # Another thread replicates, commits, and applies the
  # command, and advances lastApplied, see CommitEntry.
  await(self.lastApplied >= index)
  if self.state != "leader":
    # Deposed, don't know if command succeeded.
    return "not leader"
  return "ok"

# Handle a read request from a client for key k.
def ClientRead(k):
  if self.state != "leader": return "not leader"
  # Last committed entry's age is calculated using
  # bounded-uncertainty clock.
  if self.log[self.commitIndex].age > delta:
    return "no lease"
  # Prevent "limbo" reads.
  if self.term != self.log[self.commitIndex].term:
    if any limbo region entry affects k:
      return "key affected by limbo region"
  return self.data[k]

# When this node learns some followers have replicated
# entries up to index i, advance the commitIndex.
def CommitEntry(i):
  if self.state != "leader": return
  if not majorityAcknowledged(self.log[i]):
    return
  # Check for past-term entry < Delta old.
  # In reality this loop is optimized away, Sec. 7.
  for e in self.log:
    if e.term < self.term and e.age < delta:
      return
  self.commitIndex = max(self.commitIndex, i)
  while self.lastApplied<self.commitIndex:
    apply(self.log[self.lastApplied+1].command)
    self.lastApplied += 1
```

Fig. 2. LeaseGuard leader logic for a key-value store.

Expressed another way, once a leader has committed *any* entries, its newest committed entry is its lease. On the other hand, a new leader that *hasn't* committed entries considers its newest (possibly uncommitted) entry from the prior term as the *prior* leader's lease.

3.2 Deferred commit writes

In existing protocols, a non-leaseholder cannot accept writes. In LeaseGuard a leader can always accept writes, send them to followers, and make them fault-tolerantly durable. It simply cannot

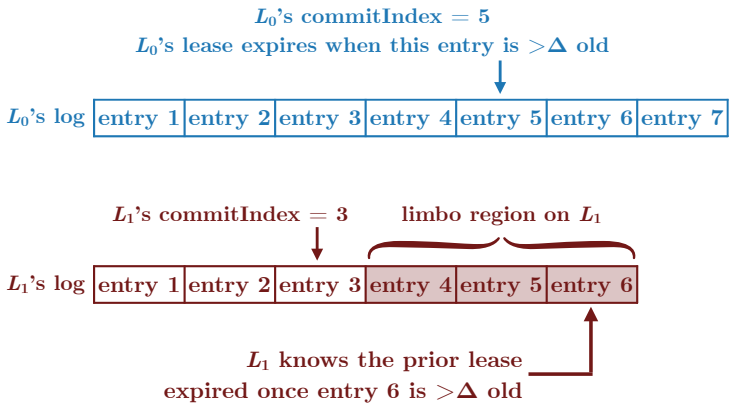


Fig. 3. Logs of old leader L_0 and new leader L_1 just after L_1 was elected, before it commits any entry in its term.

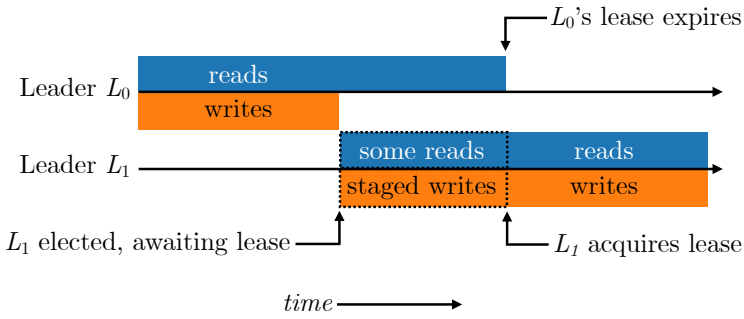


Fig. 4. Transitions in the read/write availability of leaders with LeaseGuard. While the new leader waits for a lease, it can serve some consistent reads and accept writes. Meanwhile the old leader serves reads.

commit, apply, or acknowledge writes until its latest prior-term entry (i.e., the deposed leader's lease) is more than Δ old. We call this the *deferred commit optimization*. When the prior-term entry is old enough, the leader advances its commitIndex to the index of the latest majority-replicated entry, and acknowledges all the pending writes that are now committed.

This optimization lets the leader prepare writes to be committed as soon as the old lease expires without requiring further follower communication. Even if the leader crashes or is deposed before these writes are committed, they are durable once majority-replicated, and will be committed by a future leader. By accepting writes while waiting for a lease, the new leader avoids being overwhelmed by a thundering herd of writes the moment its lease begins.

3.3 Inherited lease reads

LeaseGuard allows multiple leaders to read simultaneously, significantly enhancing read availability during leadership transitions (Figure 4). Consider a leader L_0 that has been deposed by L_1. L_0 may still believe it is the leader and can serve reads while its last committed entry e is $< \Delta$ old. L_1 (and any future leader) is guaranteed to have e (Leader Completeness) and will not advance its own commitIndex until e is more than Δ old. Thus L_0 has the latest committed data in the replica set while its lease is valid.

In LeaseGuard, we propose that L_1 can also serve linearizable reads with the lease it inherited from L_0. L_1 has all the entries L_0 committed. It knows L_0 cannot commit any more entries in its leadership term, because when L_1 won the election it increased a majority of the nodes' terms and prevented L_0 from committing any more entries [42]. Since L_1 has the latest committed data, this might seem to guarantee that L_1 can execute linearizable reads, but there is an additional kink: we need to take the "limbo region" of the log into account.

Consider Figure 3. L_1 has been elected in a later term than L_0, but L_1 has not yet committed any entries, and its commitIndex lags that of L_0. We define L_1's "limbo region" as entries in L_1's log with indexes greater than L_1's commitIndex, up to the last entry in L_1's log when it was elected. L_1 does not know where in the range [3,6] L_0's commitIndex falls. If L_1 acts pessimistically and executes client reads from its commitIndex, it may return older data than L_0 (if L_0 is alive), violating linearizability. If L_1 acts optimistically and applies all commands through index 6, it may return newer data than L_0, which also violates linearizability.

Therefore, LeaseGuard adds an additional check: a node executes reads **only if they are unaffected by writes in the limbo region**. This way both L_0 and L_1 can serve linearizable reads. Once L_0's lease expires, L_1 commits an entry and the limbo region disappears. For a simple key-value store, it is self-evident whether a write in the limbo region affects some point query or range query. Databases that support more complex queries require specialized algorithms. For example, if a database supports multi-version concurrency control (MVCC), the leader could execute a query on each version of its data corresponding to each entry in the limbo region, and reject the query if any results are unequal. To make this practical is the subject of future research.

4 Correctness

In this section, we show that LeaseGuard guarantees linearizability. Section 4.1 describes Raft's guarantees. Section 4.2 shows that LeaseGuard is correct, assuming perfect clocks. We consider clock uncertainty in Section 4.3, and discuss why correctness is preserved under reconfiguration in Section 4.4.

4.1 Raft guarantees

We rely on the following guarantees, which we quote from [42]:
Leader Append-Only: a leader never overwrites or deletes entries in its log; it only appends new entries.
Log Matching: if two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index.
Leader Completeness: if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms.
State Machine Safety: if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index.

4.2 Correctness with perfect clocks

A concurrent computation is linearizable if it is equivalent to a sequential computation that preserves the real-time ordering of operations; that is, each operation appears to take effect at some instant between its invocation and acknowledgment [22]. Raft without LeaseGuard permits multiple leaders at a time, but it is nevertheless linearizable. Raft guarantees that only the highest-term leader can commit writes, that it commits and applies each write between its invocation and acknowledgment, and that writes are applied sequentially in the same order on all nodes. To ensure linearizable reads, Raft performs a quorum check for each read to ensure the leader executing the read is the highest-term leader.

LeaseGuard alters Raft's behavior in two ways. First, a leader may wait before committing recent writes and acknowledging them to the client (*deferred commit write*, Section 3.2). Raft has no real-time bound on when a leader commits an entry after it is majority-acknowledged, so this delay cannot break Raft's linearizability guarantee.

Second, a leader can execute a read without contacting other nodes (*inherited lease reads*, Section 3.3). We sketch a proof that LeaseGuard guarantees Read-Your-Writes, and therefore guarantees linearizability.

Theorem 1 (Read-Your-Writes). *If a leader L_1 in term t_1 commits and applies write w, represented by log entry e_1 with index i_1, a later read r must observe w's effect.*

To prove this, we show that r observes w's effect in three cases:

Case 1: r is sent to L_1 while it is still the leader in term t_1. We know that w was applied on L_1, and Raft prevents undoing writes, so Theorem 1 holds. If L_1 becomes a leader again in a future term, we call it L_2 and handle it in Case 3.

Case 2: r is sent to a deposed leader L_0 with term $t_0 < t_1$. L_1 has L_0's last committed entry e_0 in its log by Leader Completeness. LeaseGuard requires L_1 to wait until all prior-term entries in its log are more than Δ old before committing an entry in t_1. Since L_1 has committed e_1, we know that e_0 is more than Δ old. LeaseGuard requires L_0 to have a committed entry less than Δ old in order to read (Section 3.3), but e_0 is L_0's last committed entry and it is older than Δ; thus L_0 rejects r and upholds Theorem 1.

Case 3: r is sent to leader L_2 with term $t_2 > t_1$. We subdivide this case according to the state of L_2's log and commitIndex.

Case 3.1: L_2's commitIndex is $i_0 \leq i_1$ and L_2 has no limbo region. By Leader Completeness, e_1 is in L_2's log, thus since it has no limbo region $i_0 = i_1$. By State Machine Safety, L_2 has applied or will eventually apply w. Raft requires L_2 to wait until its lastApplied reaches the commitIndex L_2 had when r arrived, before executing r (see ClientRead in Figure 2). So L_2 applies w before executing r, and satisfies Theorem 1.

Case 3.2: L_2's commitIndex is $i_0 \leq i_1$ and it has a limbo region, meaning that it has not committed an entry in its term. L_2's limbo region spans from its commitIndex i_0 to its last prior-term log index i_2, where $i_0 \leq i_1 \leq i_2$, so the limbo region includes w. According to the limbo read rule in Section 3.3, L_2 will not execute r if it is affected by a command in the limbo region. L_2 will reject r if it is affected by w, otherwise execute r; either way the client cannot observe that w has not been executed, so Theorem 1 is upheld.

Case 3.3: L_2's commitIndex is $i_2 > i_1$. Thus it has no limbo region, and it must eventually apply w (State Machine Safety). It must apply w some time before it executes r, satisfying Theorem 1.

Theorem 1 holds in all cases: read r observes write w's effect regardless of whether r is sent to the leader that committed w, an earlier-term leader, or a later-term leader (with or without a limbo region or a committed entry in its term).

Now we show that Theorem 1 implies that reads reflect the latest committed state among all leaders in the replica set. Note that it is theoretically possible for a replica set with $2f + 1$ nodes to have $f + 1$ leaders simultaneously: each received $f + 1$ votes from itself and its f non-leader peers. In real systems, more than two leaders is practically unheard of.

Theorem 2 (Read At Highest commitIndex). *For the set of nodes $\mathcal{L}$ whose state is "leader" at some moment, let i_{max} be the greatest commitIndex over $\mathcal{L}$. Any read r observes the effects of all commands c_i associated with all entries e_i that were ever committed by any leader, for all $i \leq i_{max}$, for the value of i_{max} at some moment between r's invocation and acknowledgment.*

Proof: Suppose r observes some earlier version of the state machine before the entry with index i_{max} was applied. There must be a write w associated with the i_{max} entry, whose effect r does not observe. This contradicts Theorem 1, thus r cannot observe the state machine at an earlier version.

Raft guarantees that the sequence of state machine versions at sequential commitIndexes is a linearizable history, and that the latest commitIndex in the replica set increases monotonically. Since LeaseGuard ensures that each read observes the latest commitIndex in the replica set, LeaseGuard preserves linearizability.

4.3 Correctness with clock uncertainty

In Case 2 above, a node must determine whether a time interval was recorded on another node more than Δ ago. This requires clocks with correctly measured uncertainty bounds on each node, such that t_1 is more than Δ old if *intervalNow()* returns t_2 and $t_1.latest + \Delta < t_2.earliest$. Such clocks have become available in the cloud over the last two years (Section 2.2). If a clock's bounds are wrong, i.e. if the true time is outside the interval [*earliest*, *latest*] for the entire duration between the invocation and completion of a call to *intervalNow()*, then multiple leaders may think they are leaseholders at once. In that case a deposed leader may execute reads while a higher-term leader commits writes. Reads from the deposed leader will violate Read-Your-Writes, and thus violate linearizability. Inherited lease reads require correct clock bounds!

4.4 Correctness with reconfigurations

Raft has two protocols for "reconfiguration", i.e. adding and removing replica set nodes: joint consensus, or single-node changes [46]. Both protocols uphold all the Raft guarantees we rely on, including Leader Completeness and State Machine Safety. With **joint consensus**, a replica set with configuration C_{old} switches to configuration C_{new} by adding and removing arbitrary members. The replica set transitions through a period when all entries must be committed by a majority of both C_{old} and C_{new}, then it retires C_{old} and makes all subsequent decisions with a majority of C_{new} only. With **single-node changes**, the replica set can either add one node or remove one node per reconfiguration, and there is no transitional period. The protocol is safe because a majority of the old and new configuration always overlaps if they differ by one server. This overlapping-majorities property preserves the Raft guarantees on which LeaseGuard depends.

5 Extensions and optimizations

5.1 Automatic lease extension

LeaseGuard extends leases automatically through write operations, but a lease expires after Δ duration with no writes. When writes are rare, the leader can write a no-op to reestablish its lease whenever needed to serve a read, or (to avoid cold starts) proactively write a no-op to maintain its lease. It is also possible to change Δ dynamically, e.g. to increase Δ to reduce the frequency of no-ops. This is out of our current scope. For planned leader transitions (e.g. for rolling upgrades and other maintenance tasks), the outgoing leader can relinquish its lease by committing an "end-lease" entry as its final act of leadership. The next leader can execute reads and writes without restriction.

5.2 Choosing election timeout and Δ

LeaseGuard allows the election timeout ET and lease duration Δ to be tuned independently. If ET is a fixed number, it is best to set $\Delta = ET$ (Figure 5). If $\Delta < ET$ and writes are rare, the leader must extend its lease more often without any benefit from the short Δ: the long election timeout causes unavailability after a crash, regardless of leases. If $\Delta > ET$, there is a period after an election when the new leader has no lease. LeaseGuard tolerates this better than prior protocols, because

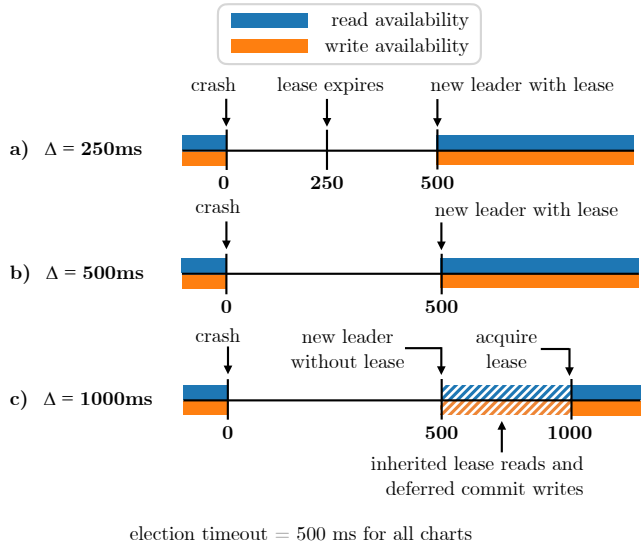


Fig. 5. Effect of lease duration on availability in LeaseGuard. Election timeout = Δ is usually optimal.

the elected leader can still serve linearizable reads using the inherited lease optimization and perform deferred commit writes (see evaluations in Sections 6 and 7). Regardless of *ET*, a leader might be deposed at any time, perhaps by a human operator or a failure detector. We anticipate that LeaseGuard will enable deployment of more aggressive failure detectors, since it improves availability after an election.

5.3 Leases without bounded-uncertainty clocks

Most of LeaseGuard can be implemented using local *timers* with bounded drift rates; bounded-uncertainty clocks are not required. All nodes must know some value ϵ, the maximum amount that their clocks can gain or lose while measuring a duration of Δ. Whenever a leader creates or a follower replicates an entry, it starts a timer to track that entry's age. The leader's timer starts as soon as it creates an entry, before any followers can replicate the entry, so the leader's timer expires before any follower's timer for that entry. LeaseGuard's rules can be revised to use timers instead of time intervals: a leader can advance its commitIndex or serve reads so long as its last entry in any previous term is $> \Delta + \epsilon$ old, and the last committed entry in its own term is $< \Delta - \epsilon$ old.

Timers suffice for LeaseGuard with deferred commit writes, but not for inherited lease reads: Suppose a leader L_1 is deposed by L_2, which commits no entries before it is deposed by L_3. Due to a network partition, L_2 is unaware of L_3's election; L_3 is elected by a quorum that does not include L_2. Now L_2 and L_3 are both leaders at the same time. They may disagree on when L_1's lease expires, based on when they last replicated an L_1 entry when they were followers. This disagreement can lead to a linearizability violation: L_3 believes L_1's lease has expired, so L_3 commits entries, while L_2 thinks the lease it inherited from L_1 is still valid, so L_2 serves inherited lease reads from stale data, which does not reflect L_3's writes. Thus, systems must have access to clocks with bounded uncertainty to implement inherited lease reads. With such clocks, L_1 stores a time interval in each entry, and there is no ambiguity about whether its entries are $> \Delta$ old.

5.4 Lessons learned

Our experience offers several lessons for the design of replicated databases. First, we found that Raft's *Leader Completeness* property provides a surprisingly powerful foundation for reasoning about leases: by tying lease validity to log replication rather than to ad-hoc timers or heartbeats, we eliminate entire classes of gray-failure and faux-leader bugs that plagued production systems.

Second, our work reinforced the value of formal methods not only as a tool for verification, but also as a tool for design discovery: both our deferred commit and inherited lease optimizations emerged from exploring corner cases in our TLA+ model. At the same time, our early iterations reminded us that formal tools are not foolproof. Our first specification missed the "limbo-region" read problem because we modeled using large atomic actions that hid the bug. This experience taught us to approach even successful model checks with skepticism: as Lamport puts it, "always be suspicious of success."

Finally, reasoning through these bugs and correctness arguments led us to view LeaseGuard through the lens of *knowledge and common knowledge in distributed systems* [21]. We are exploring how to make this epistemic-reasoning approach more practical for designing distributed database protocols.

6 Simulation

We wrote a Raft simulation in Python. Simulation is helpful for quickly exploring performance characteristics, which are impossible to analyze with a TLA+ specification, and difficult to measure cleanly with a real system [6, 7, 12]. Our simulation is at a level of abstraction somewhere between our specification and our implementation. It includes I/O latency, periodic no-ops for lease extension, and other details we omit in TLA+. On the other hand, it is far more abstract than the C++ implementation: it has no storage layer, network protocol, or query language. The simulation runs in a single process. It represents nodes as Python objects that pass messages. Time, clock error bounds, thread scheduling, and network delays are all simulated. For nondeterministic events such as checking an imperfect clock, or sending a message with unpredictable latency, we choose appropriate probability distributions and produce values with a pseudorandom number generator (PRNG). For a given seed and set of parameters, the PRNG produces the same sequence of values, thus the simulator executes the same events. We carefully engineered this reproducibility to ease debugging and analysis. The simulator expresses the LeaseGuard algorithm more simply and directly than the implementation, providing a useful reference (in addition to the TLA+ spec and C++ code) for implementers. We implemented two consistency mechanisms in our Raft simulator: quorum checks as described in the Raft paper, and LeaseGuard.

6.1 Simulator architecture

The simulator comprises these files (about 1000 lines of code, excluding whitespace and comments):

`simulate.py`: An event loop. Callbacks are scheduled to run at times in the future. The event loop finds the callback with the earliest deadline, advances the simulated clock to exactly that deadline, executes the callback (which may schedule more callbacks), then repeats. Atop the loop we layered an API with tasks, futures, and coroutines, similar to Python's standard `asyncio`. Coroutines simulate processes and threads, but they are deterministic, with the same scheduling and interleaving each time the simulation runs with a given seed.

`lease_guard.py`: Implements Raft with various consistency mechanisms, including LeaseGuard. Any number of nodes can form a replica set (our experiments use three nodes). Nodes communicate

via message-passing, with random network delays. Each node has an independent, bounded-uncertainty clock. Nodes expose two commands to clients: `write(key, value)` permits a client to append `value` to the append-only list associated with `key`, and `read(key)` returns the values appended to this list, in order. We use append-only lists because they are ideal for checking that our algorithm upholds linearizability [32] (Section 6.2).

`client.py`: Simulates a client. Any number of clients can concurrently communicate with the Raft leader. Each client appends one unique value to one key, or reads the list of values for one key. For our experiments we implement workload generators that launch many concurrent clients. These workload generators are *open loop*: they start requests at a fixed rate regardless of the response latency.

`params.py`: Contains parameters that control every aspect of the Raft and LeaseGuard protocols, e.g. whether leases are enabled, which optimizations are enabled, the election timeout, and lease duration. The module also includes parameters that control the simulation, e.g. maximum clock error bounds, the network latency mean and variance, the number of clients and their arrival rate.

`prob.py`: Produces pseudorandom numbers with various probability distributions.

`run_with_params.py`: Reads parameters from `params.py`, configures a replica set, waits for it to elect a leader, concurrently runs a large number of clients, gathers statistics such as average read/write latency, and checks that the execution was linearizable.

6.2 Linearizability checking

We implemented a linearizability checker in Python to catch bugs in `lease_raft.py`, i.e. nonconformance with the TLA+ specification. Each run of the simulator compiles a history of operations. A history is a sequence of `ClientLogEntry` objects with these fields:

- `op_type`: ListAppend or Read.
- `start_ts`: True time the client started the operation.
- `execution_ts`: True time the operation completed.
- `end_ts`: True time the client received the reply.
- `key`: The key that was read from or written to.
- `value`: The appended value for ListAppend, or the returned list of values for Read.
- `success`: Whether the operation succeeded.

Linearizability checking in general is NP-complete [18] because it requires evaluating all serializations that respect real-time order. Our simulator has the advantage of omniscience: it knows the true time when all client and server events occur. Each ListAppend entry's `execution_ts` records when the write was committed on the leader, and each Read entry's `execution_ts` marks when it was executed. To check linearizability, we check that each operation's execution time falls between its start and end times, then sort operations by execution time, and verify that Reads observe all preceding ListAppends for the same key. In some cases we still must consider multiple serializations: (1) If multiple ListAppends share the same exact execution time, we test all orderings. The history is linearizable if at least one ordering preserves ListAppend and Read semantics. (2) If a ListAppend fails from the client's perspective, it may have succeeded on the server, e.g., if the leader was deposed after replication but before responding. We check linearizability under both possibilities: the ListAppend either succeeded after it started or never succeeded. Despite these permutations, checking simulated histories with hundreds of operations usually completes in a few seconds.

6.3 Simulated experiment design

We perform simulated experiments to evaluate the following:
Q1 (Latency): How do several consistency mechanisms affect the latency of linearizable reads and writes?
Q2 (Availability): How do several consistency mechanisms affect availability after a leader crashes?
Q3 (Skewness): How does workload skewness affect read availability after a leader crashes?

6.4 Latency simulation

For Q1, we simulate a range of one-way network latencies between server nodes, to explore the effects of latencies within and across regions. We use a lognormal distribution with means from 1-10ms and with variance equal to the mean, to simulate the latencies we would observe within and across regions. Client-server latency is zero. There are 50 clients, half do a single Read operation and half do a single ListAppend. Clients arrive according to a Poisson process, with an average of 100ms between arrivals.

Figure 8(a) shows how network latency affects read and write 90th-percentile latency in three configurations. The **inconsistent** configuration does not ensure Read-Your-Writes consistency during elections. Write latency is determined by the time it takes the leader to replicate the write to at least one follower and receive acknowledgment. Read latency is zero in our simulation, since the leader serves reads without communicating with followers. The **quorum** configuration uses Raft's default consistency mechanism: the leader exchanges messages with a majority of nodes for each read, therefore read latency is determined by the round-trip time between the leader and at least one follower. The 90th-percentile write latency is higher in this configuration (6% higher with 10ms one-way network latency), because replication competes with quorum checks for I/O. The **LeaseGuard** configuration uses our protocol, which enables instant linearizable reads served locally. Writes are as fast as the "inconsistent" configuration, since replication no longer competes with reads for I/O. LeaseGuard improves consistency with no overhead.

6.5 Availability simulation

For Q2, we used network latency parameters derived from AWS same-subnet statistics (191 μs mean, 391 μs variance) [23]. An open-loop workload generator starts an operation every 300µs. One-third of operations write 1 kilobyte to a random key, two-thirds read a random key. Keys are uniformly randomly selected from a list of 1000. The election timeout ET is 500ms. (This value is within the range used by other Raft implementations; [42] proposes timeouts from 12ms to 300ms. Production systems use larger timeouts, typically 1 to 10 seconds to avoid spurious elections, e.g. CockroachDB's default is 9 seconds and TiDB Placement Driver's is 10.) We set lease duration Δ to 1 second (twice ET) to observe leader-transition availability effects. In practice, a system that uses a constant ET to detect leader failures should set $\Delta = ET$, see Section 5.2. However, a sophisticated system might use a dynamic failure detector that calls for an election whenever it suspects the leader, even when its lease is a long time from expiring. Setting $\Delta = 2ET$ is a simple way to examine this scenario. We study the effects of various consistency mechanisms on availability (Figure 7(a)). In each experiment we create a 3-node replica set and begin executing the workload. After 500ms, the leader crashes; 500ms later another leader is elected, and 500ms after that the old leader's lease expires.

Inconsistent: No mechanism ensures Read-Your-Writes. After the first leader crashes, all reads fail. Once the new leader is elected, they resume at their normal throughput instantly. Writes also fail during the interregnum between the old and new leader. After the election, clients resume writing to the new leader, but the leader has not yet committed any writes. There is a short delay

while the surviving follower discovers the new leader and begins replicating. Write throughput then appears to spike as the follower catches up and acknowledges recent log entries, and the leader commits them and acknowledges them to the client. This option has good availability but does not guarantee linearizability.

Quorum: The leader communicates with a majority of nodes (itself and at least one follower) for each read/write. All operations fail during the interregnum, and reads and writes are both delayed a short time after the election. Since reads and writes compete for I/O, they both show a throughput spike after the election, but the spike is lower than the write spike in the "inconsistent" scenario. Once the new leader has cleared its replication backlog, throughput returns to the steady state. This option has good availability, at the cost of a lower throughput ceiling and higher latency (Section 6.4).

Log-based lease: This is Raft with LeaseGuard, but without inherited read leases or deferred-commit writes. After the election, reads and writes fail until 1500ms when the new leader acquires its lease. (An implementer could choose to retry operations instead of fail-fast.)

Defer commit: The new leader begins to accept writes, create log entries, and replicate them as soon as it is elected, while waiting for the old lease to expire. However, it does not advance the commit index until it acquires a lease at 1500ms. These writes are all acknowledged near-simultaneously when the leader fast-forwards its commit index, so write throughput goes off the chart and the replica set returns to steady state quickly. The deferred commit write optimization allows the system to stabilize more quickly than with unoptimized leases.

LeaseGuard: Our lease protocol with all optimizations enabled. The new leader can serve linearizable reads as soon as it is elected, without waiting for the old lease to expire. There are few writes in progress when the leader dies (since write interarrival times are large compared to network latency), so the new leader has a small limbo region which does not significantly interfere with reads. The inherited lease read optimization restores read availability sooner after an election.

In summary: When $\Delta > ET$, leader leases inevitably hurt availability. But this experiment shows that LeaseGuard's two optimizations, inherited read leases and deferred-commit writes, help the replica set return to steady state more quickly after a leader failure.

6.6 Skewness simulation

We use Q2 parameters with Zipfian skewness a ranging from 0 to 2 across 1000 keys. At $a = 0$, keys have equal probability of being read or written; at $a = 2$, the hottest key accounts for 61% of operations. As in Q2, the leader crashes and a new one is elected. It can use an inherited lease to read any data written by the prior leader, except for data affected by log entries in the limbo region (Section 3.3). Higher skew makes reads more likely to conflict with limbo region entries. We place 100 log entries into the limbo region and measure the simulated read throughput on the new leader (Figure 6). The number 100 is chosen to stress-test the protocol for skewness effects; in reality the number of limbo entries is determined by workload intensity and the network latency among servers. Once the inherited lease expires, all reads are permitted.

7 LogCabin evaluation

7.1 Experiment design

We implemented LeaseGuard in the LogCabin codebase, the reference implementation of Raft. Prior to our work, LogCabin did not use leases: instead, LogCabin leaders ensured consistent reads with a quorum check, as described in the Raft paper. LogCabin implements a key-value database atop the Raft log. Its codebase is 27,000 lines of C++ excluding tests and comments; LeaseGuard required adding or changing 1600 lines. To execute a sufficiently intense open-loop workload from

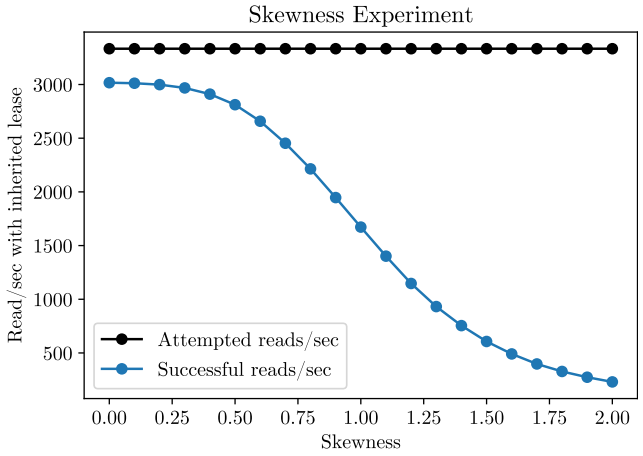


Fig. 6. Effect of workload skewness on read throughput on the new leader while it awaits a lease (simulation). More skew means fewer consistent reads are permitted. Throughput recovers once the leader acquires a lease.

a single client node required enhancing the LogCabin client with an async API, another 650 added or changed lines. The client changes were more difficult than the server changes, but necessary to ensure a valid benchmark, where the client's offered load always matched our intended intensity, no matter whether the servers experienced high latency or hit a throughput ceiling [45].

We ignore efficiency in our abstract description of the LeaseGuard algorithm, but our C++ implementation is reasonably optimized. For example, in our pseudocode for CommitEntry (Figure 2), the leader determines if it has any previous-term log entry $< \Delta$ old by iterating its whole log. In C++, the leader caches a `lastEntryInPreviousTermIndex` variable, which always points to the latest log entry in the last term, so the CommitEntry check is constant-time. Our pseudocode for ClientRead does not specify how the leader checks if any limbo region entry affects the queried key. In C++, we optimize this check, while preserving Raft's layer separation: the consensus layer manages the log and elections, while the state machine layer manages the keys and values, and the two layers are ignorant of each other's domains. In our implementation, when a node is elected, its consensus layer calls a new method `StateMachine::setLimboRegion(vector<Entry>)`. This method receives a list of entries in the limbo region and stores an `unordered_set<string>` of keys affected by limbo region entries. While the node waits to establish a lease, the state machine efficiently rejects queries involving keys in this set. When the node acquires a lease, the consensus layer calls `setLimboRegion` again with an empty vector, indicating that the state machine can read all keys freely.

We also implemented in LogCabin the "Ongaro lease" protocol, for comparison. By "Ongaro lease" we mean the mechanism proposed by Ongaro in [41] §6.4.1. Ongaro sketches the mechanism in a few sentences and has not implemented it, so we have inferred some details and implemented the highest-availability version of the protocol we can imagine. This protocol depends on the Raft rule that a follower will not vote for a candidate if it has heard from a leader less than ET ago. In our implementation of Ongaro leases, the leader tracks the start time of each AppendEntries message it sends to any follower. If the follower replies, the leader updates a local variable s_i, which is the start time of the last successful AppendEntries sent to node i. The leader considers its own s_i to be the current time. If a majority of s_i values are less than ET old, then the leader has a lease, because it knows a majority of followers have heard from it recently and therefore have not voted

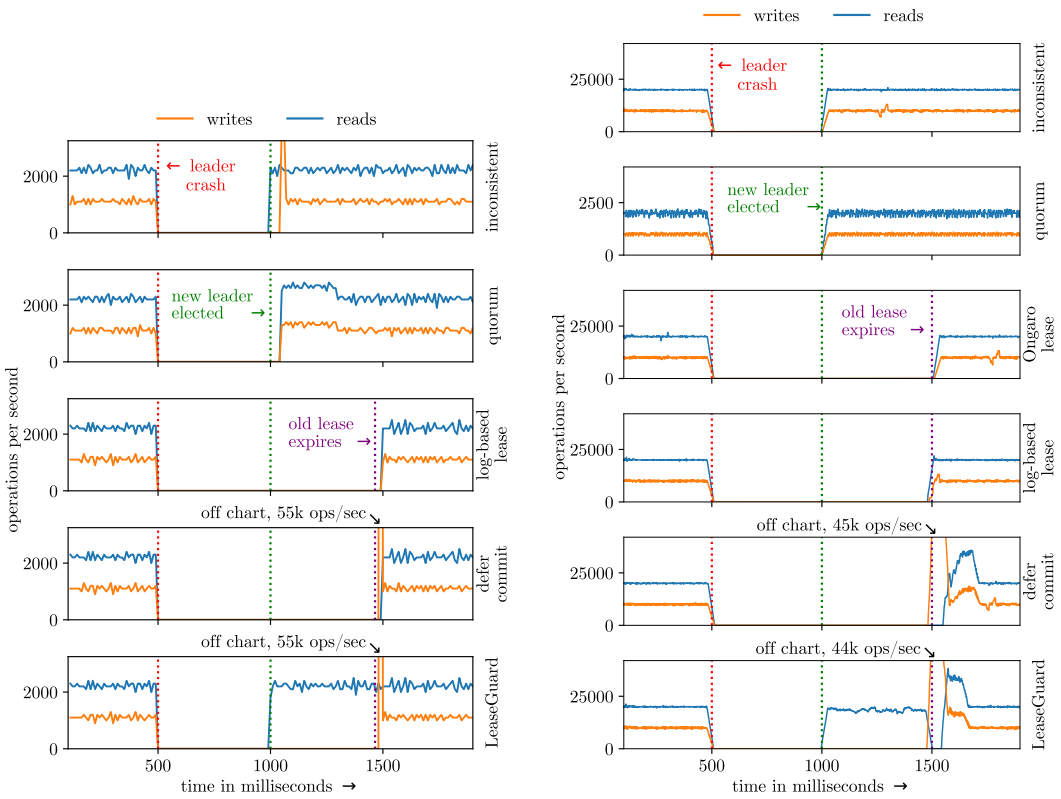


Fig. 7. Availability with Δ = 1 second, ET = 500 ms. Just after failover, LeaseGuard's deferred commit optimization increases write throughput, and its inherited lease reads improve read availability.

for another node in an election. A leader with a lease runs queries without communicating with followers. In Ongaro's description the lease is shortened slightly to compensate for clock drift; we did not implement this safety check. The Ongaro lease protocol lacks the defer commit and inherited lease read optimizations, and does not allow ET to be different from Δ.

In our experiments each node is an EC2 m7g.xlarge in the us-east-1 region. The single client and three servers are in the same placement group. Ping times among them average 155µs. Starting in 2024, servers of this instance type use AWS TimeSync by default, but for maximum clock precision we installed the Precision Time Protocol kernel module, recompiled Amazon's Elastic Network Adapter with physical hardware clock support, and installed Amazon's open-source clock-bound daemon [27]. The clock-bound API provides time intervals spanning less than 100µs (clock error less than 50µs). We used our modified LogCabin implementation to replicate two of our three simulated experiments, plus a fourth:

Q1 (Latency): How do leases affect the latency of linearizable reads and writes?
Q2 (Availability): How do leases, inherited leases, and deferred-commit writes affect availability after a leader crashes?
Q4 (Scalability): What is the max throughput with various consistency mechanisms?

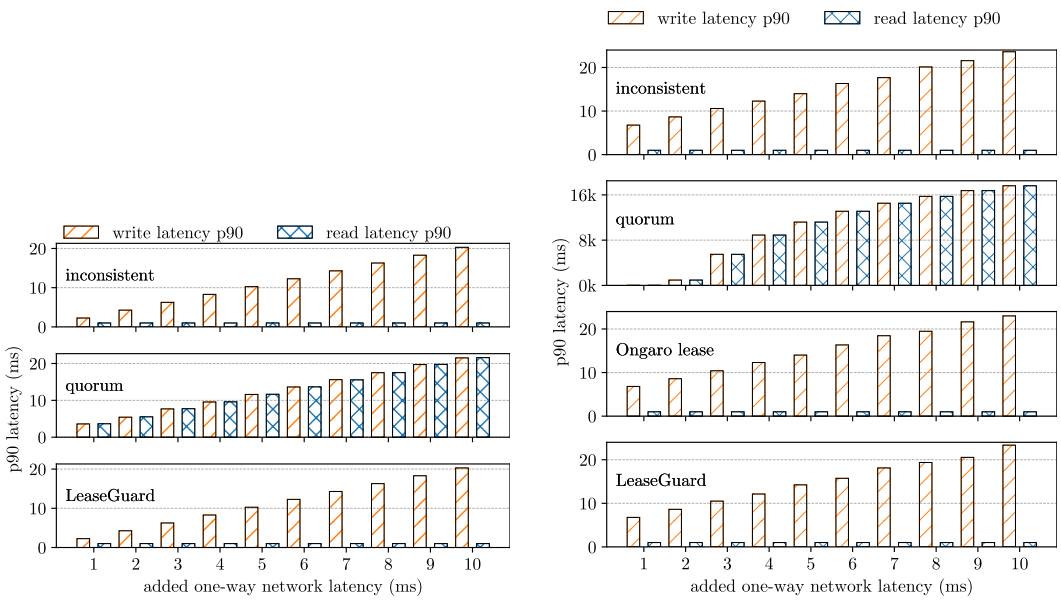


Fig. 8. Effect of network latency on read/write latency (milliseconds). Quorum reads increase latency; LeaseGuard makes consistent reads as fast as inconsistent reads.

7.2 Latency

For Q1, the client executes an open-loop workload. One third of operations write a 1-kilobyte value each, and two-thirds of operations read a 1-kilobyte value. The client starts an operation every 300 microseconds and measures its round-trip time. We use "tc", the Linux traffic control utility, to add one-way latency from 1 to 10ms between server nodes. This range covers the latencies we would observe between servers within and across regions. Client-server latency is unaffected. Figure 8(b) shows how network latency affects 90^{th}-percentile read and write latency, in the same three configurations as we used for the simulated experiment, plus "Ongaro lease". For all configurations, writes require leader-follower communication and write latency is affected by latency between servers. In the "inconsistent", "Ongaro lease", and "LeaseGuard" configurations, reads require no communication with the followers, so 90^{th}-percentile latency is in the hundreds of microseconds. Performance is similar to the simulation's. In the "quorum" configuration, reads require leader-follower communication, and they compete with writes for I/O, harming latency for both. A small increase in network latency causes a huge increase in operation latency due to queueing effects, which we do not observe in our simulation. LeaseGuard permits consistent reads and writes with zero overhead compared to Ongaro leases or the inconsistent configuration.

7.3 Availability

As we did in the simulated experiment, we set lease duration Δ to 1 second (twice ET) to observe the effect of various consistency mechanisms on availability after an election (Figure 7(b)). Ongaro leases do not have a separate Δ parameter so we set ET to 1 second for Ongaro leases. An open-loop workload generator starts 30 operations per millisecond. One-third of operations write 1 kilobyte to a random key, two-thirds read a random key. Keys are randomly selected from a list of 1000

according to a Zipf distribution with $a = 0.5$, so the hottest key is chosen 1.6% of the time. We test four consistency configurations:

Inconsistent: No mechanism ensures linearizability. The same as in the simulation, throughput recovers soon after the election.

Quorum: LogCabin's default consistency mechanism. Note that the Y axis for this chart is one tenth as high as the others: quorum checks dramatically harm throughput. LogCabin shows a much worse impact from quorum checks than our simulation does.

Ongaro Lease: The leader serves reads if a majority of successful RPCs began less than ET ago. Throughput is restored when a new leader is elected.

Log-based lease: Our lease protocol with no optimizations. As in the simulation, the system is unavailable after the election until the old leader's lease expires.

Defer commit: LeaseGuard with our write optimization. We see a write spike when the new leader acknowledges all the writes it had buffered while waiting for a lease. This spike is not as vertical as in the simulation, but still goes off the chart to 90 writes per millisecond. The avalanche of acks briefly slows read throughput, which then spikes and recovers. The deferred commit write optimization reduces failed writes and allows the system to stabilize quickly after an election.

LeaseGuard: Our lease protocol with both optimizations (deferred commit writes and inherited lease reads). Unlike our simulation, our actual test produces a significant limbo region of 37 possibly-committed log entries on the new leader that prevent some linearizable reads. Therefore read throughput is slightly lower while the new leader waits for a lease: the client attempts 10,000 reads during this half-second period, but only 9,930 succeed. Nevertheless, the inherited lease read optimization improves read availability while the new leader waits for a lease.

7.4 Scalability

We subjected LogCabin to increasing load with our async client, starting at 5000 operations per second and stopping at 60,000 operations per second, or once latency increased over 100 ms. As in our latency experiment, the client reads and writes 1 kilobyte values to uniformly-distributed random keys. Figure 9 shows LogCabin's throughput and latency with various consistency mechanisms and write ratios. Its throughput with LeaseGuard matches or exceeds that of Ongaro leases for each workload.

8 Related work

8.1 Leases

Leases were originally introduced for distributed file cache consistency [19, 35]. Consensus with leader leases was implemented for MultiPaxos in [9, 10, 14]. Unlike Raft, MultiPaxos does not guarantee Leader Completeness, so it is incompatible with LeaseGuard's simplification ("the log is the lease") and its read optimization.

A Raft lease protocol was described in [41], but not specified in detail or built into LogCabin, Raft's reference implementation. In that protocol, the leader starts a timer, sends a heartbeat message to all nodes, and acquires a lease once a majority replies. The lease lasts until the timer, plus some clock-drift compensation, exceeds some timeout. The leader extends its lease periodically. Followers promise not to vote for a new leader until the last known lease expires. Compared to LeaseGuard, this protocol delays elections and thus harms read and write availability. It is prone to gray failures, where a leader can maintain its lease although it cannot write log entries. It also risks a thundering herd of requests overwhelming the new leader once it acquires a lease.

TiKV uses a separate lease management service called the TiDB Placement Driver, requiring additional infrastructure and communication [50]. The TiDB Placement Driver includes a Raft

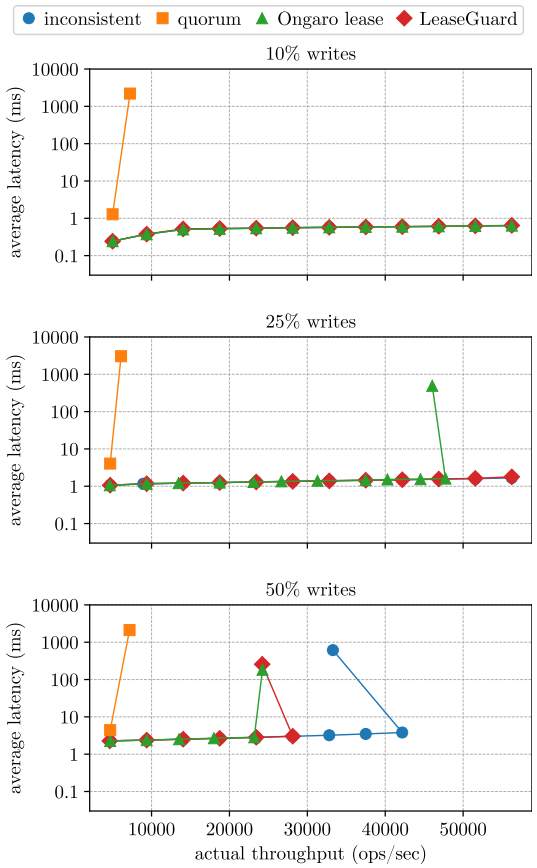


Fig. 9. LogCabin scalability with various consistency mechanisms. LeaseGuard is more scalable than quorum check or Ongaro leases and nearly equals the inconsistent configuration.

implementation with a 10-second leader lease, resulting in a 10-second outage after any leader failure [34]. Spanner implements Paxos with leases using tightly synchronized clocks [14]. In both protocols, a node that knows of a lease does not call an election or cast a vote until that lease expires. In contrast, LeaseGuard does not delay elections or modify the election protocol at all.

YugabyteDB's implementation of Raft leases [44] allows leader election to overlap with an outstanding lease, like ours. But during elections, voters must tell the candidate explicitly about existing leases. LeaseGuard is simpler. It avoids the need for an explicit lease-learning mechanism by inferring the lease from the log; this technique was partly inspired by [13].

In CockroachDB as described in [49], replicas hold leases on data ranges, and leaseholders need not be leaders. A replica acquires/extends its lease on a data range by writing a special lease-acquisition log entry. In LeaseGuard, *every* log entry acquires/extends the lease, obviating the need for a special entry type. CockroachDB lacks our improvements to read and write availability.

In CockroachDB as described in [49], replicas hold leases on data ranges. The leaseholder is typically the Raft leader, though not always. A replica acquires or extends its lease on a data range by writing a special lease-acquisition log entry. In LeaseGuard, *every* log entry acquires/extends

the lease, obviating the need for a special entry type. CockroachDB lacks our improvements to read and write availability.

8.2 Other mechanisms for consistent reads

Ongaro proposed a mechanism for ensuring causal consistency in Raft in [41] §6.4.1: the leader includes its lastApplied index with each reply, and the client remembers this index and includes it with each request. If a leader receives a request with a lastApplied greater than its own, it waits until its lastApplied catches up before executing the request. This guarantees causal consistency because Raft ensures that all servers apply the same commands in the same order (State Machine Safety). We implemented this mechanism in MongoDB years ago, but customers dislike it. It requires an application to use the same client for all related requests, which is onerous in a scaled-out cloud application. Besides, some applications require linearizability; causal consistency is relatively weak.

Prior research investigated linearizable reads from *followers* to ease the leader's load. In the Paxos Quorum Lease protocol [38], followers have leases, so an individual follower can provide linearizable reads without communicating with its peers. The leader needs acknowledgment from each leaseholding follower for each write, so the loss of any single follower can harm write availability. Other work [3, 11, 56] ensures linearizable follower reads by contacting a quorum of followers. While these relieve the leader of the read load, they require communication with a majority, and cannot match the efficiency of leader leases.

For strongly-consistent transactional follower reads, Arora et al. [3] detect write conflicts using CockroachDB's *write intents*. A write intent is a per-key marker that records any uncommitted update's timestamp, value, and transaction metadata. A client sends a quorum read to a majority, and followers return data and the timestamp for each key. A follower that encounters a write intent, however, returns only a timestamp without data, signaling an uncommitted update. The client checks whether the highest timestamp in the quorum lacks data, and if so it retries its query. This protocol improves steady-state read scalability by safely offloading reads to followers. It bears a resemblance to LeaseGuard's limbo region: both are ways to determine which keys are safe to read. But our limbo region's purpose is to guarantee non-transactional consistent local reads on the *leader* just after an election, not the follower; and the limbo region is a range of log entries, not a marker on a key.

BUC-Raft [53] proposes two Raft optimizations to reduce latency. In the original Raft protocol, a command must be committed, then applied to the leader's data, before it is acknowledged to the client. BUC-Raft introduces Commit Return (CR), which lets the leader acknowledge writes as soon as they are committed, without awaiting application. This introduces a risk of reading stale data, which BUC-Raft solves with Read Acceleration (RA). RA executes each query on the leader's current data, then patches up the query result with any relevant committed-but-unapplied commands, to ensure freshness. BUC-Raft claims to preserve linearizability, but this implicitly requires leader leases, since BUC-Raft has no mechanism of its own to prevent clients querying a deposed leader. RA resembles our limbo-read mechanism, but with fundamental differences. RA aims to reduce steady-state read latency caused by the lag between committing and applying writes, whereas our limbo reads improve availability just after an election. BUC-Raft's CR and our deferred-commit optimization solve different problems with different techniques: CR acknowledges committed commands sooner and shifts risk to the read path, whereas LeaseGuard accepts writes sooner but withholds acknowledgment until they are committed. Overall, BUC-Raft targets steady-leader performance, whereas LeaseGuard ensures consistency and maximizes availability during leader transitions. The approaches are orthogonal and can be composed.

Recent work by Katsarakis et al. [30] proves the LAW impossibility: *in linearizable asynchronous read/write registers tolerating a single crash, no reads can be local.* They then introduce almost-local

reads (ALRs): eager and lazy schemes that execute reads locally but add a lightweight "sync" per batch (or piggyback on a timely write) so the result is linearizable under asynchrony. LeaseGuard, on the other hand, assumes partial synchrony and the availability of bounded-uncertainty clocks. LeaseGuard provides linearizable reads from the leader with no communication overhead or waiting.

8.3 Bounded-uncertainty clocks

The use of bounded-uncertainty clocks in distributed systems has become more common recently. Spanner [14] famously uses Google's proprietary time infrastructure to enforce consistency, and CockroachDB [49] uses a similar mechanism adapted for public clouds. Accord [48] and Detock [39] use time to order conflicting transactions. YugabyteDB [43], Aurora Limitless [47], and Aurora DSQL [8] use AWS TimeSync to enforce consistency with high-precision clocks.

9 Conclusion

We presented a novel leader lease protocol for the Raft consensus algorithm that relies on bounded-uncertainty clocks and the guarantees provided by the Raft log. LeaseGuard simplifies the design and implementation of leader leases while minimizing read and write unavailability during elections. We provide TLA+ specifications and correctness proofs for our protocol. Through analysis, simulation, and benchmarking, we demonstrated LeaseGuard's availability and efficiency benefits.

References

[1] 2020. *IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems.* doi:10.1109/IEEESTD.2020.9120376

[2] Amazon Web Services. 2023. Amazon Time Sync Service now supports microsecond-accurate time. https://aws.amazon.com/about-aws/whats-new/2023/11/amazon-time-sync-service-microsecond-accurate-time/ Accessed: 2024-07-14.

[3] Vaibhav Arora, Tanuj Mittal, Divyakant Agrawal, Amr El Abbadi, Xun Xue, Yanan Zhi, and Jianfeng Zhu. 2017. Leader or Majority: Why have one when you can have both? Improving Read Scalability in Raft-like consensus protocols. In *9th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2017, Santa Clara, CA, USA, July 10-11, 2017*, Eyal de Lara and Swaminathan Sundararaman (Eds.). USENIX Association, Santa Clara, CA, USA, 6 pages. https://www.usenix.org/conference/hotcloud17/program/presentation/arora

[4] RethinkDB Authors. 2024. RethinkDB, the open-source database for the realtime web. https://rethinkdb.com/

[5] boringhello. 2024. etcd-io/raft Issue 166: Question about LeaseRead. https://github.com/etcd-io/raft/issues/166

[6] Marc Brooker. 2022. Formal Methods Only Solve Half My Problems. https://brooker.co.za/blog/2022/06/02/formal.html

[7] Marc Brooker. 2022. Simple Simulations for System Builders. https://brooker.co.za/blog/2022/04/11/simulation.html

[8] Marc Brooker. 2024. DSQL Vignette: Reads and Compute. https://brooker.co.za/blog/2024/12/04/inside-dsql.html Accessed: 2025-03-09.

[9] Michael Burrows. 2006. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, Brian N. Bershad and Jeffrey C. Mogul (Eds.). USENIX Association, Seattle, WA, USA, 335–350. http://www.usenix.org/events/osdi06/tech/burrows.html

[10] Tushar Deepak Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos made live: an engineering perspective. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC 2007, Portland, Oregon, USA, August 12-15, 2007*, Indranil Gupta and Roger Wattenhofer (Eds.). ACM, Portland, Oregon, USA, 398–407. doi:10.1145/1281100.1281103

[11] 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 2019, Renton, WA, USA, July 8-9, 2019*, Daniel Peek and Gala Yadgar (Eds.). USENIX Association, Renton, WA, USA, 7 pages. https://www.usenix.org/conference/hotstorage19/presentation/charapko

[12] Audrey Cheng, Shu Liu, Melissa Pan, Zhifei Li, Bowen Wang, Alex Krentsel, Tian Xia, Mert Cemri, Jongseok Park, Shuo Yang, Jeff Chen, Lakshya Agrawal, Aditya Desai, Jiarong Xing, Koushik Sen, Matei Zaharia, and Ion Stoica. 2025. Barbarians at the Gate: How AI is Upending Systems Research. arXiv:2510.06189 [cs] doi:10.48550/arXiv.2510.06189

[13] Archie Cobbs. 2017. Lock and leadership election in Raft. https://groups.google.com/g/raft-dev/c/oO0NfgUVrjg/m/R1BnnbNcAwAJ

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

[15] Mike Curtiss. [n. d.]. Why you should pick strong consistency, whenever possible. https://cloud.google.com/blog/products/databases/why-you-should-pick-strong-consistency-whenever-possible. Accessed:2025-07-15.

[16] etcd Authors. 2024. etcd, a distributed, reliable key-value store for the most critical data of a distributed system. https://etcd.io/

[17] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. 2018. Exploiting a Natural Network Effect for Scalable, Fine-grained Clock Synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*, Sujata Banerjee and Srinivasan Seshan (Eds.). USENIX Association, Renton, WA, USA, 81–94. https://www.usenix.org/conference/nsdi18/presentation/geng

[18] Phillip B. Gibbons and Ephraim Korach. 1997. Testing Shared Memories. *SIAM J. Comput.* 26, 4 (1997), 1208–1244. arXiv:https://doi.org/10.1137/S0097539794279614 doi:10.1137/S0097539794279614

[19] Cary G. Gray and David R. Cheriton. 1989. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the Twelfth ACM Symposium on Operating System Principles, SOSP 1989, The Wigwam, Litchfield Park, Arizona, USA, December 3-6, 1989*, Gregory R. Andrews (Ed.). ACM, Litchfield Park, Arizona, USA, 202–210. doi:10.1145/74850.74870

[20] Jim Gray. 1978. Notes on Data Base Operating Systems. In *Operating Systems, An Advanced Course (Lecture Notes in Computer Science, Vol. 60)*, Michael J. Flynn, Jim Gray, Anita K. Jones, Klaus Lagally, Holger Opderbeck, Gerald J. Popek, Brian Randell, Jerome H. Saltzer, and Hans-Rüdiger Wiehle (Eds.). Springer, 393–481. doi:10.1007/3-540-08755-9_9

[21] Joseph Y. Halpern and Yoram Moses. 2000. Knowledge and common knowledge in a distributed environment. *CoRR* cs.DC/0006009 (2000). https://arxiv.org/abs/cs/0006009

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

[23] Owen Hilyard, Bocheng Cui, Marielle Webster, Abishek Bangalore Muralikrishna, and Aleksey Charapko. 2024. Cloudy Forecast: How Predictable is Communication Latency in the Cloud?. In *33rd International Conference on Computer Communications and Networks, ICCCN 2024, Kailua-Kona, HI, USA, July 29-31, 2024*. IEEE, Kailua-Kona, HI, USA, 1–9. doi:10.1109/ICCCN61486.2024.10637631

[24] Heidi Howard and Richard Mortier. 2020. Paxos vs Raft: have we reached consensus on distributed consensus?. In *7th Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2020, Heraklion, Greece, April 27, 2020*, Alan D. Fekete and Martin Kleppmann (Eds.). ACM, Heraklion, Greece, 8:1–8:9. doi:10.1145/3380787.3393681

[25] 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 (2020), 3072–3084. doi:10.14778/3415478.3415535

[26] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. 2022. Metastable Failures in the Wild. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, Marcos K. Aguilera and Hakim Weatherspoon (Eds.). USENIX Association, 73–90. https://www.usenix.org/conference/osdi22/presentation/huang-lexiang

[27] Amazon.com Inc. 2021. clock-bound. https://github.com/aws/clock-bound Accessed: 2025-02-11.

[28] Clockwork Inc. 2022. clockwork. https://www.clockwork.io/ Accessed: 2025-02-12.

[29] HashiCorp Inc. [n. d.]. Consul. https://www.consul.io. Accessed: 2025-07-05.

[30] Antonios Katsarakis, Vasilis Gavrielatos, Emmanouil Giortamis, Pramod Bhatotia, Aleksandar Dragojevic, Boris Grot, Vijay Nagarajan, and Panagiota Fatourou. 2025. The LAW Theorem: Local Reads and Linearizable Asynchronous Replication. *Proceedings of the VLDB Endowment* 18, 9 (2025), 2831–2845.

[31] Kyle Kingsbury. [n. d.]. Jepsen: etcd and Consul. https://aphyr.com/posts/316-jepsen-etcd-and-consul. Accessed:2025-07-10.

[32] Kyle Kingsbury. 2021. Elle: Finding Isolation Violations in Real-World Databases. In *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*, Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen (Eds.). ACM, Italy, 7. doi:10.1145/3465084.3467483

[33] Leslie Lamport. 2002. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, Boston, MA, US. http://research.microsoft.com/users/lamport/tla/book.html

[34] Yishuai Li, Yunfeng Zhu, Chao Shi, Guanhua Zhang, Jianzhong Wang, and Xiaolu Zhang. 2024. Timestamp as a Service, not an Oracle. *Proc. VLDB Endow.* 17, 5 (2024), 994–1006. doi:10.14778/3641204.3641210

[35] Barbara Liskov. 1993. Practical Uses of Synchronized Clocks in Distributed Systems. *Distributed Comput.* 6, 4 (1993), 211–219. doi:10.1007/BF02242709

[36] Barbara Liskov and James Cowling. 2012. *Viewstamped Replication Revisited.* Tech. Rep. MIT-CSAIL-TR-2012-021. MIT.

[37] Keith Marzullo and Susan Owicki. 1983. Maintaining the time in a distributed system. 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, 295–305. doi:10.1145/800221.806730

[38] 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, November 3-5, 2014*, Ed Lazowska, Doug Terry, Remzi H. Arpaci-Dusseau, and Johannes Gehrke (Eds.). ACM, Seattle, WA, US, 22:1–22:13. doi:10.1145/2670979.2671001

[39] Cuong DT Nguyen, Johann K Miller, and Daniel J Abadi. 2023. Detock: High performance multi-region transactions at scale. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.

[40] Oleg Obleukhov and Ahmad Byagowi. 2022. How Precision Time Protocol is being deployed at Meta. https://engineering.fb.com/2022/11/21/production-engineering/precision-time-protocol-at-meta/ Accessed: 2025-03-07.

[41] Diego Ongaro. 2014. *Consensus: bridging theory and practice.* Ph. D. Dissertation. Stanford University, USA. https://searchworks.stanford.edu/view/10608105

[42] Diego Ongaro and John K. Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference, USENIX ATC 2014, Philadelphia, PA, USA, June 19-20, 2014*, Garth Gibson and Nickolai Zeldovich (Eds.). USENIX Association, Philadelphia, PA, USA, 305–319. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[43] Franck Pachot. 2024. Achieving Precise Clock Synchronization on AWS. https://www.yugabyte.com/blog/aws-clock-synchronization/ Accessed: 2025-03-09.

[44] Karthik Ranganathan. 2019. Low Latency Reads in Geo-Distributed SQL with Raft Leader Leases. https://www.yugabyte.com/blog/low-latency-reads-in-geo-distributed-sql-with-raft-leader-leases/

[45] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. 2006. Open Versus Closed: A Cautionary Tale. In *3rd Symposium on Networked Systems Design and Implementation (NSDI 2006), May 8-10, 2007, San Jose, California, USA, Proceedings*, Larry L. Peterson and Timothy Roscoe (Eds.). USENIX. http://www.usenix.org/events/nsdi06/tech/schroeder.html

[46] William Schultz, Siyuan Zhou, Ian Dardik, and Stavros Tripakis. 2021. Design and Analysis of a Logless Dynamic Reconfiguration Protocol. In *25th International Conference on Principles of Distributed Systems, OPODIS 2021, December 13-15, 2021, Strasbourg, France (LIPIcs, Vol. 217)*, Quentin Bramas, Vincent Gramoli, and Alessia Milani (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Strasbourg, France, 26:1–26:16. doi:10.4230/LIPICS.OPODIS.2021.26

[47] Anum Jang Sher and David Wein. 2024. Achieving scale with Amazon Aurora PostgreSQL Limitless Database. https://www.youtube.com/watch?v=pUqVCK7Ggh0 Accessed: 2025-03-09.

[48] Benedict Elliot Smith, Tony Zhang, Blake Eggleston, and Scott Andreas. 2022. CEP-15: Fast General Purpose Transactions. (2022). https://cwiki.apache.org/confluence/download/attachments/188744725/Accord.pdf?version=1&modificationDate=1630847737000&api=v2

[49] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, Portland, OR, USA, 1493–1509. doi:10.1145/3318464.3386134

[50] Siddon Tang. 2018. How TiKV Uses "Lease Read" to Guarantee High Performances, Strong Consistency and Linearizability. https://www.pingcap.com/blog/lease-read/

[51] YugabyteDB Team. 2021. YugabyteDB: cloud native distributed SQL database for mission-critical applications. https://www.yugabyte.com/

[52] Jelle van den Hooff. 2016. Make leader leases actual leases. https://github.com/hashicorp/raft/issues/108

[53] Yangyang Wang, Zikai Wang, Yunpeng Chai, and Xin Wang. 2023. Rethink the linearizability constraints of Raft for distributed systems. *IEEE Transactions on Knowledge and Data Engineering* 35, 11 (2023), 11815–11829.

[54] 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, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, Peter Robinson and Faith Ellen (Eds.). ACM, Toronto, ON, Canada, 445–454. doi:10.1145/3293611.3331595

[55] Todd Warszawski and Peter Bailis. 2017. ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 5–20. doi:10.1145/3035918.3064037

[56] Jianjun Zheng, Qian Lin, Jiatao Xu, Cheng Wei, Chuwei Zeng, Pingan Yang, and Yunfan Zhang. 2017. PaxosStore: High-availability Storage Made Practical in WeChat. *Proc. VLDB Endow.* 10, 12 (2017), 1730–1741. doi:10.14778/3137765.3137778

[57] Siyuan Zhou and Shuai Mu. 2021. Fault-Tolerant Replication with Pull-Based Consensus in MongoDB. In *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*, James Mickens and Renata Teixeira (Eds.). USENIX Association, Berkeley, CA, US, 687–703. https://www.usenix.org/conference/nsdi21/presentation/zhou

Received July 2025; revised October 2025; accepted November 2025