LoLKV: The Logless, Linearizable, RDMA-based Key-Value Storage System

Ahmed Alquraan and Sreeharsha Udayashankar, *University of Waterloo;* Virendra Marathe, *Oracle Labs;* Bernard Wong and Samer Al-Kiswany, *University of Waterloo*

https://www.usenix.org/conference/nsdi24/presentation/alquraan

This paper is included in the Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation.

April 16–18, 2024 • Santa Clara, CA, USA

978-1-939133-39-7

Open access to the Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

LoLKV: The Logless, Linearizable, RDMA-based Key-Value Storage System

Ahmed Alquraan[*], *Sreeharsha Udayashankar*[*], *Virendra Marathe*[#], *Bernard Wong*[*], *Samer Al-Kiswany*[*]
[*]*University of Waterloo, Canada*
[#]*Oracle Labs, USA*

Abstract

We present LoLKV, a novel logless replicated key-value storage system. LoLKV follows a fundamentally different approach for designing a linearizable key-value storage system compared to state-of-the-art systems. LoLKV forgoes the classical log-based design and uses lock-free approach to allow multiple threads to concurrently update objects. It presents a novel leader election and consolidation approach to handle complex failure scenarios. LoLKV's followers are passive, reducing their overall CPU usage. Our evaluation shows that LoLKV achieves 1.7–10× higher throughput and 20–92% lower tail latency than other state-of-the-art low-latency key-value stores.

1 Introduction

Online services, such as financial services [1, 2] and interactive applications [3], have strict tail latency requirements in the microsecond ranges [4]. Systems supporting these services must achieve high throughput at a low tail latency. A central category of systems supporting this class of services is replicated and strongly consistent key-value stores [5, 6, 7].

State-of-the-art low-latency key-value stores such as APUS [5], DARE [6], Mu [7], and uKharon-KV[8] use RDMA-based consensus protocols, which replicate data across multiple nodes in a few microseconds. While these systems achieve acceptable tail latencies under low load, our evaluation (Section 6) shows that these systems are unable to maintain these low tail latencies under high load. The main reason for this inefficiency is that these systems follow the classical log-based design [9, 10] for building a linearizable storage.

The classical log-based design has two steps for executing operations that update objects: *replication* and *application*. During the replication step, a leader stores and *replicates* a new operation to a replicated log. Once the operation is replicated on a majority of followers, the application step *applies* the operation to the key-value store on all followers.

The log-based design [9, 10] has three fundamental inefficiencies when considering low tail latency workloads. First, the log represents a single point of serialization, limiting concurrency and prohibiting current systems from effectively leveraging multi-core machines. To address this inefficiency, modern systems resort to *sharding* [5, 6, 7, 11, 12, 13, 14], such that each shard serves a subset of the key space. Each shard has a separate process with its own memory, threads, and followers on other nodes. Current systems use sharding for two purposes. First, to distribute shards among nodes to scale to large clusters. Second, to have multiple *active* shards per node to leverage multiple CPU cores. Unfortunately, deploying multiple active shards per node leads to inefficient memory use, especially when paired with RDMA, as each shard process has its own pinned memory region causing memory fragmentation. Furthermore, having a large number of shards complicates supporting multi-key operations and leads to lower performance for skewed workloads, in which a few shards hold popular keys.

The second fundamental inefficiency is that the classical log-based design separates data replication from application. In key-value stores this leads to an extra memory copy as new objects need to be first stored in the log [5, 6, 7] and later, when committed, copied to the key-value store.

Finally, the classical design requires all followers to re-execute every operation. This significantly increases system overhead. For instance, with a replication level of three, every operation is executed three times in the cluster.

We present the Logless Linearizable Key-Value (LoLKV) storage system. LoLKV follows a fundamentally different design approach compared to the state-of-the-art systems. LoLKV forgoes the replicated log design and avoids placing multiple active shards on a node. In LoLKV, a node may have a single active leader shard and multiple passive follower shards. The leader shard is multi-threaded and utilizes all of the node's CPU cores. The passive follower shards are not involved in processing `put` and `get` operations. The leader shard uses one-sided RDMA to replicate new objects on followers. Once the new object is replicated on a majority of nodes, the hash table is updated with a pointer to the new object.

This approach overcomes the aforementioned shortcomings. First, it uses multiple threads to utilize all CPU cores and to concurrently replicate objects. It avoids extra memory copies by only having pointers in the hash table. Finally, the leader replicates the updates to the memory and hash table of followers without requiring the follower to re-execute operations. This enables LoLKV to use all available resources on all nodes to serve client requests, leading to higher performance.

While the proposed design increases system concurrency, it introduces new challenges for leader election and data consolidation. The proposed design complicates fault tolerance because LoLKV uses multiple concurrent threads that can leave the system in an inconsistent state under leader failure scenarios. Applying insert and delete operations directly in the key-value store complicates garbage collection as well. To realize LoLKV, we design novel leader election and data consolidation protocols that identify the latest updates for each memory segment and sync a new leader with the latest

system state. In addition, LoLKV uses a replicated garbage collection approach to maintain the storage system memory in sync during garbage collection.

We implemented LoLKV and compared it to the state-of-the-art systems. Our evaluation shows that for uniform workloads, LoLKV achieves *1.69–2.9*× higher throughput and *20–55%* lower tail latency over DARE [6], and *4–10*× higher throughput and *56–92%* lower tail latency over APUS [5], Mu [7] and uKharon [8]. In addition, our evaluation shows that LoLKV achieves similar results under skewed workloads and under different read and write ratios. Our evaluation of system scalability shows that LoLKV can scale to efficiently use all system resources and achieve up to *18 million op/s* which is 4×, 7×, 10×, and 36× higher than DARE, uKharon, Mu, and APUS, respectively.

The rest of the paper is organized as follows. Section 2 discusses RDMA and RDMA-based consensus protocols. Section 3 and 4 discuss the design and implementation of LoLKV. Section 5 discusses the correctness of LoLKV. Section 6 presents the results of our evaluation. Section 7 discusses additional related work before the paper concludes in Section 8.

2 Background and Related Work

2.1 Remote Direct Memory Access

Remote direct memory access (RDMA) [15] allows a machine to directly access the memory of a remote machine without involving the remote CPU. RDMA offers a low-latency and high throughput communication mechanism as it bypasses the kernel network stack.

Applications communicate over RDMA by establishing queue pairs. Each queue pair consists of a send and a receive queue. RDMA supports two main types of operations: two-sided and one-sided operations. Two-sided operations include `Send` and `Receive` operations and involve the CPUs of both the sender and receiver. One-sided operations do not involve the CPU on the receiving side. The sender specifies the data's remote address. One-sided operations have lower latency and higher throughput compared to two-sided operations [16].

Current implementations of RDMA support three transport protocols: Reliable Connected (RC), Unreliable Connected (UC), and Unreliable Datagram (UD). In RC and UC, a connection is established between two queue pairs, one on the sender and one on the receiver, which communicate exclusively with each other. RC guarantees that messages are delivered at most once, in order, and without corruption. In a UD protocol, one queue pair can communicate with one or more queue pairs. UD supports only two-sided operations.

In LoLKV, we use one-sided `Writes` over RC for data replication and two-sided operations over UD for client communication.

2.2 RDMA-based Consensus

Several RDMA-based leader-based consensus protocols have been recently proposed. In leader-based systems, one replica acts as a leader while others are followers. The leader is responsible for processing client requests. The leader appends new operations to the log and then replicates the log entry on followers. The operation is considered committed only if it is replicated on a majority of replicas. The leader then applies the operation to the state machine and replies to the client.

DARE [6] is an in-memory RDMA-based consensus protocol that adopts the Raft protocol [9]. Replication in DARE requires 2 RTT of one-sided `Write` operations. The first set of `Writes` appends the log entry to the follower logs while the second updates their tail indices. When both `Writes` succeed on a majority of replicas, the leader updates its commit index and posts another `Write` to update the followers' commit indices. Followers check their commit index periodically to apply newly committed entries to their state machines.

APUS [5] is a Paxos-based [10] consensus protocol which uses RDMA. The leader stores new operations to its local log and then replicates log entries to follower logs using one-sided `Writes` and waits for acknowledgments. In contrast to DARE, APUS followers actively participate in replication. Each follower notifies the leader when it accepts a log entry by sending an RDMA `Write` to update the entry in the leader's log. The leader commits an entry if it is accepted by a majority of replicas. Committing an entry in APUS requires 2 RTTs.

Mu [7] is an RDMA-based consensus protocol that targets microsecond-scale applications. The leader uses an RDMA `Write` to append operations to follower logs. The operation is considered committed if it is replicated on a majority. Followers poll committed requests from the log and pass them to the application. The leader does not start replicating a log entry until the previous one is committed. Mu requires a single RTT to replicate an operation in the common case.

uKharon [8] is an RDMA-based consensus protocol optimized for microsecond-scale failure recovery. uKharon uses one-sided RDMA-based Paxos to achieve fast leader failover times. To enable one sided Paxos, uKharon's consensus engine uses an RDMA Write and a Compare-and-Swap (CAS) operation in the ACCEPT phase while Mu [7] only uses an RDMA Write. While this additional CAS improves failure recovery times, the authors mention that it causes uKharon to perform worse than Mu in failure-free scenarios. In addition, uKharon does not offer ways for failed or network-partitioned nodes to rejoin the system.

These systems have three fundamental shortcomings. Firstly, at the core of each of these systems is a replicated log. The log limits concurrency because it introduces a serialization point. New operations are inserted serially in the log. Furthermore, in Mu, a new object is replicated only when the previous object is committed.

We evaluate the overhead of having multiple threads appending to a log. Figure 1 shows the time a thread waits until it acquires the lock protecting the log in APUS and Mu. Results show that the average wait time for 8 threads is 14.9× and 38.5× higher than that with one thread for APUS and

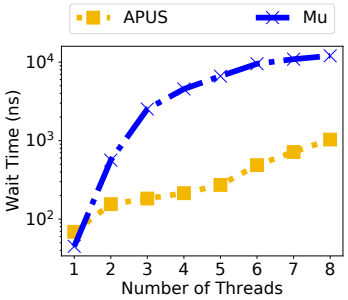

Figure 1: The average time needed to acquire the lock protecting the operation log in APUS and Mu.

Mu, respectively. Mu has a higher overhead as Mu holds onto the lock until the operation is replicated and committed, while APUS releases the lock after appending the request to the log. Given the low latency that RDMA provides, the delay imposed by the log serialization mechanism introduces a bottleneck that limits the performance of the system.

Second, the log-based design imposes an additional data copy. New objects are first inserted into the log and then copied to the key-value store. Finally, to efficiently leverage all CPU cores, multiple leader shards are deployed per node. Deploying multiple leader shards per node requires physical partitioning of memory between co-located shards, leading to inefficient memory usage. Also, sharding leads to lower performance under skewed workloads.

LoLKV adopts a fundamentally different design approach to overcome these shortcomings. LoLKV adopts a logless design that co-designs the storage and index data structures to avoid unnecessary memory copies. LoLKV deploys a single multi-threaded leader shard per node, which is able to efficiently utilize all CPU cores. It also uses novel techniques for leader election, data consolidation, and garbage collection. These design decisions lead to higher concurrency and lower system overhead.

3 LoLKV Design

LoLKV supports `get`, `put`, and `delete` requests, which read, write, and delete entire objects respectively. Figure 2 shows the architecture of LoLKV. The system consists of a set of nodes (i.e., replicas or replica set) connected using an RDMA-capable network, such as InfiniBand [15].

LoLKV adopts a leader-based replication protocol, in which one replica acts as a leader while others are followers. All client requests are handled by the leader, which runs multiple worker threads to process them. LoLKV does not partition the key space among the threads, i.e., any thread can process `put` or `get` requests for any object.

Similar to Raft [9] and Multi-paxos [17], LoLKV divides time into terms with a single leader per term. Each term has a unique `term_id` that is assigned in a strictly increasing order. When the leader fails, a new leader is elected using a leader election protocol (Section 3.4).

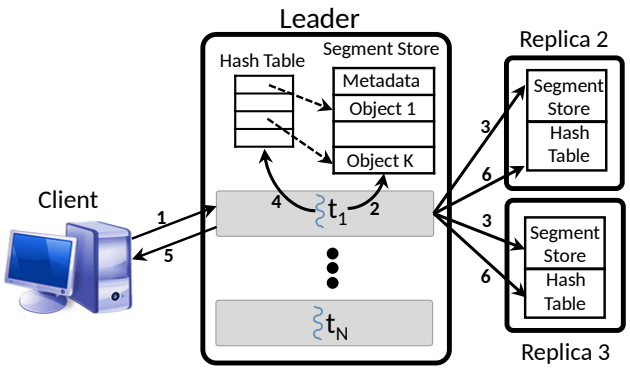

Figure 2: LoLKV Architecture.

LoLKV has two components: the *segment store*, which holds objects within memory segments; and the *hash table*, which contains pointers to these objects. To store a new value, the leader first stores the value in its local segment store (Step 2 in Figure 2) before replicating it on follower nodes (Step 3). A new value is considered committed only if it is stored in the segment store of a majority of followers. After an entry is committed, the leader updates its local hash table (Step 4) and replies to the client (Step 5). Updates to the hash table are replicated lazily to the followers (Step 6) since the object metadata stored in the segment store has enough information to recover the hash table in case of leader failure.

LoLKV guarantees linearizability at the object level; updates to the same object appear to be executed in a single global order. While updates to different objects can proceed concurrently. To manage concurrency within the segment store, each worker thread has exclusive access to a subset of the memory segments. The hash table is shared among all threads and uses CAS operations to serialize concurrent updates by multiple threads.

3.1 Segment Store

LoLKV divides allocated memory into equal sized segments which hold objects. Each segment contains a metadata section and an array of `ObjectEntries`. All objects within a single segment have the same size but objects across different segments may have different sizes. LoLKV's segment store design is inspired by the Hoard memory allocator [18].

A segment can be owned by a single worker thread at a time. The thread which owns a segment has exclusive write access to its objects. A segment can be *free* meaning it is unowned, *active* meaning it has space for new objects, or *sealed* meaning that it is full and its objects cannot be modified. As stored objects are immutable, any thread can read any object from any segment. Each thread typically owns multiple segments with different object sizes.

3.1.1 Segment Metadata

Figure 3a shows the fields of a segment's metadata:

(a) Segment Metadata (b) ObjectEntry

Figure 3: LoLKV data structures.

- `owner_id` : ID of the thread that owns the segment.
- `term_id` : Term during which the latest change to the segment metadata occurred.
- `status` : Indicates whether a segment is active or sealed.
- `seg_ver` : Updated when a thread owns a segment.
- `object_size` : Size of objects within the segment. Object size may differ between segments.
- `tail_idx` : Index of the latest inserted entry in the segment.

3.1.2 Owning a Segment

To claim ownership of a new segment, a thread traverses the segments array in the segment store to find a segment that is "free". The thread then uses a CAS operation to change the `owner_id` field of the metadata of the free segment. Following this, the thread sets the `seq_num` for all `ObjectEntries` to -1 to indicate that they are free. Then, it sets the fields of the segment metadata: `term_id` to the current term number, `status` to "active", and `tail_idx` to 0.

LoLKV commits segment ownerships one at a time on a majority of replicas. The thread owning a new segment acquires a lock, increments a counter shared across threads, copies the counter's value to the `seg_ver` field in the segment metadata, and then replicates this metadata. Once the operation is committed via replication on a majority of followers, the thread releases the lock and uses the segment to store new objects. This process guarantees that a segment's ownership is committed before it is used.

3.1.3 Sealing and Releasing a Segment

A thread seals a segment when it is full and all its objects have been applied to the hash table. A segment remains sealed until all its entries are garbage collected. After that, the thread can release the segment. To release a segment, a thread will change the `owner_id` to -1 and set the status to "free". When a segment is released, it becomes available for other threads to own. Sealing and releasing a segment are performed by changing the `status` field to "sealed" or "free", respectively.

Segment sealing and releasing operations are not replicated. If a leader fails after sealing or releasing a segment, a future leader will perform the same operations as the old leader. If it finds a full segment, it will seal the segment, and if it finds an empty segment, it will release the segment.

3.2 Hash Table

The hash table stores pointers to `ObjectEntries` stored in segments; The goal of the hash table is to implement efficient `get` operations. The size of each hash table entry is 64 bits.

A hash table entry stores the offset of an `ObjectEntry` within the segment store, to ensure portability across replicas. While segment stores may have different base addresses across replicas, an `ObjectEntry` in a segment will have the same offset on all replicas. The hash table is shared among all worker threads.

The main fields of an `ObjectEntry` are shown in Figure 3b:

- `<key,value>`: The object's key and value.
- `term_id` : Term during which this entry was written.
- `seq_num` : Operation sequence number assigned by the worker thread. Used to order all operations of a thread within a term.

3.3 Replication Protocol

The leader runs multiple worker threads to process client requests. Each request is processed by one worker thread. Worker threads are independent; each thread has its own RDMA resources (e.g., queue pairs and work completion queues).

3.3.1 Worker Thread Metadata

Each worker thread maintains the following metadata fields: `sequence_number` and `latest_operation`. The `sequence_number` is a strictly increasing counter that is used to assign a sequence number for `put` operations processed by the thread. The `latest_operation` field stores the address of the last `ObjectEntry` inserted by the thread.

The thread also maintains two metadata fields for each follower: `latest_replicated_operation` and `latest_applied_operation`. The thread uses these to identify operations not yet replicated or applied at the follower.

3.3.2 Put Requests

The processing of a `put` request has three main phases: creation, replication, and application phases.

Creation Phase. When a thread receives a `put` request, the thread creates an `ObjectEntry` and populates its fields. The thread increments its `sequence_number` and assigns it to the entry. The thread writes the object to a segment. The thread updates its `latest_operation` field and `tail_idx` field of the segment metadata to point to the newly inserted entry.

Replication Phase. The thread commits the `ObjectEntry` by replicating it on a majority of replicas. The thread replicates entries in order of `seq_num`. Threads use RC transport to replicate objects, which guarantees in-order message delivery.

Once an ObjectEntry is replicated on a majority of replicas, the request is considered committed. The thread updates its latest_replicated_operation field whenever it replicates an entry on a follower. The <term_id, seq_num> tuple specifies the order of all operations processed by a thread.

Application Phase. After an ObjectEntry is committed, the thread updates the hash table entry to point to the newly inserted object. To guarantee consistency, entries must be applied in the order specified by <term_id, seq_num> tuple.

LoLKV uses a linear probing hash table. The thread first checks if the key already exists in the hash table. The thread hashes the key *K* to determine the entry to begin probing from. The thread checks that hash table entry and all successive entries one by one. Probing terminates when the thread finds either a hash table entry pointing to an ObjectEntry of the key *K* or an empty entry. If the thread finds an entry pointing to the key *K*, the thread only updates the entry if the insert is a newer operation than the existing version (Section 3.3.3). If probing ends at an empty entry, the key is inserted into the empty entry.

As the hash table is shared among all worker threads, multiple threads may concurrently try to update the same hash table entry. To ensure correctness, hash table entries are updated atomically using CAS operations. If a CAS operation fails, the thread repeats the probing process.

After the thread applies the entry locally, it returns an acknowledgement to the client. In the background, the thread replicates the hash table updates to the followers. Periodically, the thread updates segments' tail_idx on followers.

3.3.3 Concurrent Put Requests to the Same Key

In LoLKV, multiple put requests for the same key might be replicated concurrently by different threads. As a result, an ordering mechanism is needed to ensure the correctness of the system. LoLKV uses the *incarnation array* to ensure operations are serialized. The incarnation array is an array of *K* atomic counters. LoLKV divides the key-space into groups and each group is mapped to a counter in the incarnation array. The counter is atomically incremented whenever a put operation for a key within the group is processed. This value is then stored in the incarnation field of the ObjectEntry.

Before applying an ObjectEntry to the hash table, its <term_id, incarnation> tuple is verified to ensure that it is larger than the <term_id, incarnation> of the ObjectEntry currently pointed to by the hash table. If it is not larger, the hash table update is discarded since the ObjectEntry currently pointed to by the hash table is newer.

The incarnation array is not replicated as it is only used to order writes by leader threads. When a new term starts, the new leader resets its incarnation array. The size of the incarnation array is far larger than the number of worker threads, which reduces the chance of false sharing between concurrent put operations.

3.3.4 Get Requests

In LoLKV, get requests are served locally by the leader. Any thread can process any get request for any key as a thread can read objects from all segments, even a segment it does not own. The worker thread probes the hash table in the same manner used for put requests. The thread will return a value to the client only if the hash table entry points to a valid ObjectEntry. It is safe to serve get requests using only the leader state because the leader obtains a lease [19] from a majority of replicas. During the lease, other replicas will not try to become a leader.

3.3.5 Delete Requests

Delete requests follow the same steps of the put requests with a small difference: a tombstone object is replicated in the segment store and the hash table is updated with to point to the tombstone object.

3.4 Leader Election

LoLKV divides time into terms, and each term has at most one leader. Before a replica becomes a leader, it must get the votes of a majority of replicas. Each replica can vote only once in a term. LoLKV relies on dynamically managing the state of the RC QPs to ensure that only the current leader has remote access to the system's data structures. Before accepting a vote request from a candidate, replicas revoke the old leader's access by transitioning the associated QPs into a non-operational state.

Failure detection. LoLKV employs a heartbeating mechanism to detect failures. Each replica maintains a heartbeat data structure which consists of three fields: term_id, leader_id, and counter. The counter field is incremented periodically. The leader election protocol starts when a follower suspects that the leader has failed (i.e., misses *N* heartbeats from the leader). To detect leader failure, followers read the leader's heartbeat data structure periodically via RDMA. Followers verify that the leader's counter field has changed and that the leader's ID and term_id match their data structure. If a follower cannot access the leader's heartbeat data structure or the leader's counter does not change for a specific duration, the follower assumes that the leader has failed. Similarly, the leader checks the followers heartbeat data structures periodically and steps down if it no longer has a majority of followers.

Leader election. Leader election protocols of some systems (e.g., Raft [20]) elect the most up-to-date replica as a leader. However, using this approach in LoLKV is not feasible. In LoLKV, different worker threads are independent. Different threads may replicate different requests on different replicas. That is, two committed operations could be replicated on two different majorities. As a result, after a leader failure, there could be no replica that is up-to-date for all threads.

In LoLKV, each replica scans its segments to find the the segment with the largest <term_id, seg_ver>. Replicas exchange their largest <term_id, seg_ver>. The replica with the largest <term_id, seg_ver> becomes a leader. If multiple nodes have the same <term_id, seg_ver>, the node with the smallest IP address among them becomes the leader.

Data consolidation. The elected leader may not have all the segments up-to-date. Updating segments requires identifying the most up-to-date replica for each thread. To do this, the leader sends a request to each follower asking for the highest <term_id, seq_num> of each thread. Using this information, the leader identifies its missing operations for each thread and the replicas which have these operations. The leader contacts the most up-to-date follower to get the missing operations. The leader commits any uncommited operations and applies them to the hash table.

Update Followers. During the leader election and data consolidation steps, the leader finds the highest <term_id, seg_ver> for each follower and the highest <term_id, seq_num> for all threads of the follower. The leader uses this information to complete the work of the previous leader by committing all uncommitted memory ownership operations and put operations.

Once the leader completes this process, the leader has all committed values and its segment store is up-to-date. After that, the leader starts serving new client requests.

3.5 Fault Tolerance

Follower Failure. LoLKV is designed to tolerate N replica failures given $2N + 1$ replicas. When a follower fails, the leader removes it from the active set and stops replicating operations on that replica. Hence, follower failures do not affect the safety of the system. When a follower rejoins the system, the leader first finds the highest <term_id, seg_ver> of the follower, and then it replicates all memory ownership operations that the follower misses. Then, the leader recovers the worker threads metadata for that follower (i.e., latest_replicated_operation and latest_applied_operation). After that, the leader starts replicating ObjectEntries on the follower.

Leader Failure. When the leader fails, the system will not be available until a new leader is elected and the data consolidation process completes. For any thread, it is guaranteed that at least one replica in any majority has all committed entries that are committed by the failed leader. The new leader will use the data consolidation mechanism (Section 3.4) to bring itself up-to-date. Hence, although a leader failure might affect the availability of the system, it does not affect its correctness.

Message Loss. Client requests and responses are sent over UD QPs, which do not guarantee delivery of messages. If a request or a response is lost, the client times out and resubmits the request again. Note that it is safe to process requests again since they are idempotent. On the other hand, replication is implemented on top of RDMA RC transport which offers reliable and in-order message delivery.

Data Corruption. The data structures described in Section 3 can be accessed simultaneously by both local and remote threads. As a result, the correctness of any data read must be verified. LoLKV facilitates this by augmenting each structure with verification information. ObjectEntries are augmented with a byte appended to the end of each entry. For segment metadata entries and heartbeat data structures, a counter field is stored at the beginning of the data structure and again at the end of the data structure. If a failure occurs while updating this data structure, the counter field at the beginning will not match the one at the end.

Self-verifying data structures [13] are able to detect corruptions at the remote side as well (e.g., incomplete RDMA Write) because RDMA NICs guarantee that writes are performed in an increasing address order, i.e., the verification bytes at the end are not written before other bytes in the data structure.

4 Implementation Details

4.1 Garbage Collection

The system will eventually run out of space if segments are not released by owner threads. The garbage collection process is responsible for releasing segments. A segment is not released until all ObjectEntries in the segment become invalid, i.e., they are inaccessible through the hash table.

Each thread periodically checks its sealed segments to identify any segments that should be freed. In LoLKV, a segment is freed if the ratio of valid objects in a segment drops below a configurable threshold. Before a segment is released, all its valid entries must be moved to another active segment. The owner thread moves these entries to a new segment while preserving their incarnation and term_id values in order to avoid overwriting a newer entry of the same key. The thread then inserts the objects in the hash table using the same probing mechanism. An entry in the hash table is updated with a new address only if its incarnation equals that of the moved object. Once the segment does not have any valid entries, the thread releases the segment by updating its metadata.

4.2 Reusing Tombstones

While processing a put, if during the probing step the thread reaches an empty hash table entry, this indicates that the key does not exist. During probing, the thread records the first hash table entry that points to a tombstone with a term number smaller than the current term number. If probing terminates at an empty hash table entry (i.e., the key is not found), the thread updates the hash table tombstone to point to the new ObjectEntry. If no tombstone is found, the empty hash table entry is updated to point to the new ObjectEntry. We only reuse tombstones from previous terms to avoid conflicts with delete requests in the current term.

5 Correctness

LoLKV design is based on the following assumptions; the network is unreliable and asynchronous, as packets could be dropped or take arbitrary time to arrive at the destination. There is no limit on the time a node takes to process a request. We assume a non-byzantine failure model in which nodes may stop working but will never send erroneous messages.

LoLKV guarantees linearizability at the object level. All operations for an object appear to be executed in a single global order. We used the TLA+ model checking tool [21] to verify LoLKV's correctness. We use TLA+ model checking to verify the correctness of the segment store, the hash table, the replication protocol, and the leader election. In this section we sketch out the proof of LoLKV's correctness and discuss the correctness of the segment store and the hash table. To simplify the discussion, we sketch the proof without garbage collection.

Property I. *Memory ownership is safe*, meaning if a segment is owned by a thread, the segment ownership operation will be present in all future leaders. To prove this, we use three invariants:

1. *A leader in term i commits segment ownership operations in order of the seg_ver.* When a thread owns a new segment, it acquires a lock, increments *seg_ver* and assigns it to the new segment, commits the ownership operation on a majority of followers, then releases the lock. Using a lock guarantees that segments are assigned an increasing *seg_ver* and are committed in order of *seg_ver*.
2. *For a given term i, if a follower has segment with term_id = i and seg_ver = n, it is guaranteed that it also has all segments with term_id = i and seg_ver < n.* Following the argument in the previous invariant, segment ownership operations are committed in order of *seg_ver*. Furthermore, LoLKV uses RC transport to replicate memory ownership operation. RC guarantees ordered delivery of messages. This means if a node received *seg_ver*, then it must have received all previous messages sent on that channel.
3. *The last segment ownership operation committed in term i is present in the leader of term $i+1$.* The leader election protocol elects a leader with the highest <*term_id, seg_ver*> among a majority of replicas. If the last committed ownership operation on a majority in term i has $seg\_ver = n$, then the leader of term $i+1$ will have an ownership operation with term id = i and $seg\_ver >= n$. Following invariant 1 above, the leader of term $i+1$ will have the last committed entry in term i.

Invariant 3 indicates that a leader of term i has the latest memory ownership committed in term $i-1$. Hence, invariant 2 implies that the leader of term i also has all segment ownership committed in term $i-1$. *By induction, the leader of term i has all the segment ownership committed in all previous terms.*

Property II. *The data consolidation process guarantees that a leader has all committed entries from the previous term.* Without loss of generality, we sketch the proof for a single thread here using the following invariants:

4. *The leader thread in term i replicates put operations in order of the seq_num*. To commit an operation, the leader thread increments the *seq_num*, stores the operation in the local segment store, then replicates the operations to followers.
5. *If a follower in term i receives an operation with $seq\_num = n$, then this follower has received all operations from this thread in term i with $seq\_num < n$*. This follows from invariant 4, and that we use RDMA RC transport for replication. The RC transport guarantees in-order delivery of messages.
6. *For a given thread in term i, if an operation with $seq\_num = n$ is committed, then all operations for that thread in term i with $seq\_num < n$ are committed*. If operation with $seq\_num = n$ in term i is committed, then this operation has been received by a majority of nodes. From invariant 5 above, those nodes that received $seq\_num = n$ must have received all operations in term i with $seq\_num < n$. Given that those are a majority of nodes, then all operations in term i with $seq\_num <= n$ are committed.
7. *A node that has the highest seq_num in term i among a majority of nodes has all of the committed entries in term i.* Assume that in term i, the largest *seq_num* on a majority of nodes is n, and that the last committed entry in term i is m. Given that $seq\_num = m$ is replicated on a majority, then $m <= n$. Following invariant 5 above, the node that has n also has all operations with $seq\_num < n$, including m. Following invariant 6, it has all committed entries in term i.

The newly elected leader has the highest term number on a majority of nodes, but may not have all committed entries from the previous term. Before processing new requests, the leader runs the data consolidation process. For each thread, data consolidation looks for the highest <*term_id, seq_num*> on a majority of replicas. If a follower has a *seq_num* higher than the *seq_num* of the leader thread, then the consolidation process updates the leader with all missing operations. At the end of this process, for a given thread, the leader has all operations from the previous term with the highest *seq_num* among a majority of nodes. Following invariant 7 above, the leader has all committed entries in that term for that thread, verifying property II.

Property III. *LoLKV guarantees if an object is committed by a leader it will be present in the segment store of all future leaders.* From property II above, if a leader commits an operation in term i, the data consolidation process guarantees that this operation will exist in the segment store in the leader of term $i+1$. By induction, a committed entry will exist in the

segment store of all future leaders.

5.1 Correctness of the Hash Table

We do not need to replicate updates to the hash table entries for correctness. The object store has all the information needed to recover the hash table on a new leader. LoLKV threads lazily replicate updates to the hash table in the order of `seq_num` only to optimize leader recovery.

We discuss safety when multiple threads try to update the same slot in the hash table under two scenarios: concurrency between threads updating the same slot with different keys and concurrency between threads updating the same slot with the same key.

The hash table uses CAS to update its slots. If a thread finds an empty slot in the table and its CAS operation fails to update that slot, the thread repeats the probing operation to find the target slot for the new object.

Property IV. *LoLKV applies updates to the hash table following the order specified by the* `incarnation`. If two threads are processing a put request for the same key, one of them will insert the update first in the table using CAS. While probing, the second thread will find that the key already exists, but since the object was inserted by a different thread, we cannot use the `seq_num` to order these updates. For this scenario, we rely on the `incarnation` field. When processing a `put` request, a thread atomically increments the incarnation array to get a unique increasing `incarnation` for its `put` operation. We use this number to order operations on the same key. If a thread finds the object in the table with `incarnation` higher than the `incarnation` of its `put` operation, it discards its `put` operation, otherwise it updates the hash table.

Property V. *In LoLKV, A* `get` *request returns the latest committed object for the target key.* In LoLKV, `get` requests are served locally by the leader without contacting the followers. To serve a `get` request, a worker thread probes the hash table in the same manner used for a `put` request. If the thread finds the target key in the table, the thread reads the object from the object store. The thread verifies that the hash table slot was not modified while the thread is reading from the object store by verifying that the slot still contains the same object store address. If the slot has a different address, the thread repeats the process and reads the newer object of the key. Since a key can have at most one slot in the hash table, and since hash table slots point only to committed objects, the thread is guaranteed to return the latest committed object for the key.

Property VI. *LoLKV guarantees linearizability at the object level.* Properties I, II, and III guarantee that if a leader commits an entry in the segment store, the entry will be present in the segment store of all future leaders. Multiple threads may concurrently commit entries for the same key. Property IV shows how LoLKV uses the `incarnation` to create a global order of updates to the same key. Finally, Property V guarantees that `get` operations return the latest committed entry. These properties guarantee that LoLKV is linearizable for `put` and `get` operations per key.

6 Evaluation

We evaluate the performance of LoLKV and compare its latency and throughput against the state-of-the-art alternatives using uniform and skewed workload distributions.

Testbed. We conducted the experiments using a 12-node cluster on Cloudlab Utah [22]. Each machine has a Xeon E5-2450 8-core CPU with hyperthreading, 16GB of RAM, and a Mellanox Dual port FDR CX3 adapter. Machines are connected using a 56 Gbps Infiniband [15] network.

Alternatives. We compare LoLKV against DARE [6], APUS [5], Mu [7], and uKharon [8]. DARE and uKharon implementations come bundled with an in-memory key-value store. For APUS and Mu, we implemented an in-memory key-value store following the same design as LoLKV's in-memory key-value store. Newly inserted keys are stored in the APUS or Mu logs and then copied to the segment store during the apply phase. We also evaluated LogCabin [20], a strongly-consistent key-value store based on Raft [9]. LogCabin uses TCP/IP stack for communication. Our results with write-only workload show that LogCabin has a throughput of $12,500\ ops$, which is at least two orders of magnitude lower than other RDMA-based systems, and its latency is around $1.5ms$, which is three orders of magnitude higher latency than other RDMA-based systems. We also evaluated HERD [12], an unreplicated key-value store that uses a mixture of RDMA Writes and RDMA Send and Receive for communication with clients. Results show that HERD achieves $4\times$, $8\times$, $8\times$, and $50\times$ higher throughput compared to LoLKV, DARE, Mu, and APUS, respectively. This performance difference is mainly because HERD is an unreplicated key-value store, while other systems implement a replicated and linearizable key-value store. We omit adding the results of LogCabin and HERD to the figures for clarity.

Experiment Configuration. We run the systems in a multi-sharded setting, in which multiple processes of a system are deployed. Each process represents a replica of a single shard. We benchmark each system and select the number of shards and configurations that maximize its performance. We used a replication factor of three in all our experiments. All replicas of all shards were distributed among the same three nodes, while other nodes are used to run clients. Unless otherwise specified, we deploy the leader replicas of all shards on one node. However, we evaluate the scenario when the leaders are equally distributed among nodes in Section 6.6.

Shard Configuration. We run LoLKV with 8 threads, DARE with 8 shards, APUS with 7 shards, and Mu and uKharon with 4 shards. APUS does not scale to 8 shards as it runs a background process to establish RDMA multicast communication paths. Mu does not scale to 8 shards as each shard uses 4 threads; one thread performs log replication while 3 threads perform heart beating, monitoring, and permissions management. uKharon also does not scale to 8 shards as each

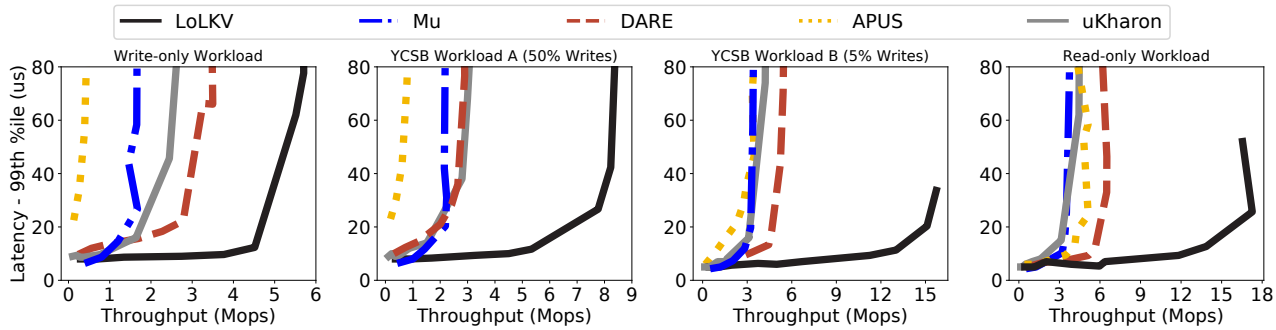

Figure 4: Throughput and tail latency using uniform YCSB workloads.

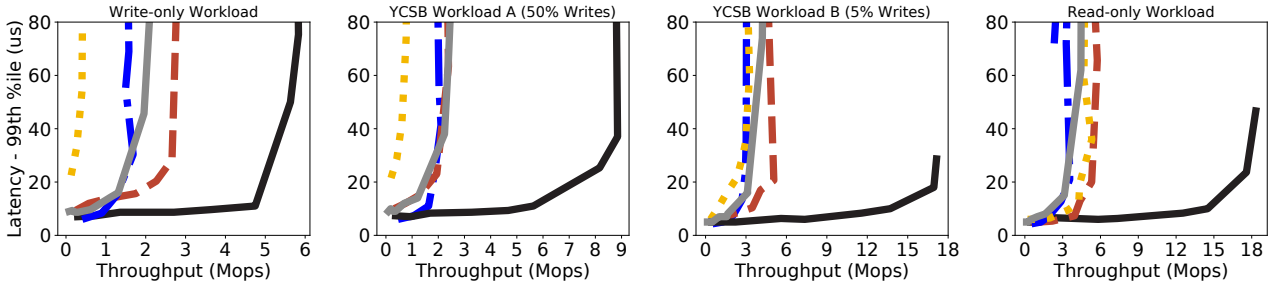

Figure 5: Throughput and tail latency using skewed YCSB workloads.

shard uses two threads, one for processing client requests and replication and one for monitoring and detecting failures.

Multi-shard deployment of DARE, APUS, Mu, and uKharon requires sharding the key space into partitions, with all requests to keys within a partition being served by the same replica set. In our experiments with these systems, a client picks the right shard for a request based on the hash of the key. On the other hand, as LoLKV does not have a fixed assignment of keys to threads, its clients send requests to threads in a round-robin fashion.

Workload. We used the YCSB benchmark [23] in our experiments with both uniform and skewed workloads. The skewed workload follows the Zipf distribution with a skewness parameter of 0.99. We experimented with 100,000, 1 million, and 10 million keys. As the results obtained in all these cases are similar, we present the results with 1 million keys. The key and value sizes are 24 and 64 bytes respectively. We report the averages from running 10 trials for each experiment. The standard deviation of all our experiments is below 5%.

6.1 Performance Evaluation

We compare the throughput and latency of all systems using the YCSB workloads with uniform and skewed key popularity distributions (Figure 4 and Figure 5).

For uniform workloads (Figure 4), LoLKV achieves *1.7–2.9*× higher throughput and *20–55%* lower latency compared to DARE. LoLKV also achieves *4–10*× higher throughput and *56–92%* lower latency compared to uKharon, Mu, and APUS. Figure 5 shows that LoLKV's performance is not affected by workload skewness. On the other hand, DARE, uKharon, Mu, and APUS achieve 14%, 16%, 6%, and 18% lower throughput with the skewed write-only workload compared to their throughput with uniform workload.

LoLKV outperforms other alternatives for several reasons. First, LoLKV requires only one RDMA Write operation to replicate an object. Second, LoLKV combines the replication phase and the apply phase by replicating operations directly to the object store, avoiding an extra memory operation to copy the data from the operation log to the object store.

APUS has the lowest performance because replication in APUS requires two RDMA Write operations: one from the leader to followers to replicate log entries and one from followers to the leader to indicate the acceptance of a log entry. Hence, in APUS, the CPUs of the followers are involved in the replication process, which reduces throughput and increases latency. Replication in Mu and uKharon requires only one RDMA write operation from the leader to the followers, which results in higher throughput and lower latency compared to APUS. However, Mu and uKharon do not employ batching in the replication process, which imposes long queuing delays under heavy workloads. DARE achieves better performance than APUS and Mu as it employs batching. However, DARE requires two RDMA Write operations to replicate a batch of operations: the first operation replicates log entries on followers and the second one updates the tail index of the followers.

LoLKV outperforms other systems even under read-heavy workloads due to two reasons. First, even with a small percentage of `put` requests, the overhead of replicating and applying operations limits the performance of `get` requests. Second, APUS, Mu, and uKharon are multi-threaded systems but not all threads are utilized to serve client requests. As a result, given the same amount of resources, LoLKV and DARE can

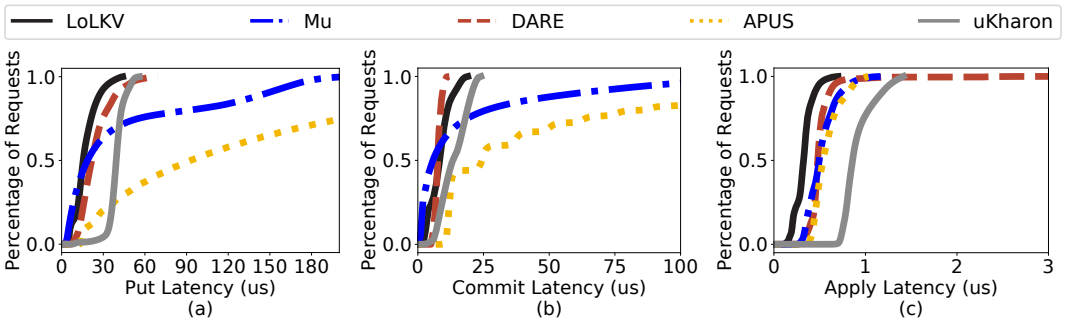

Figure 6: Latency CDF with a uniform write-only workload (a) `put` latency. (b) commit latency. (c) apply latency.

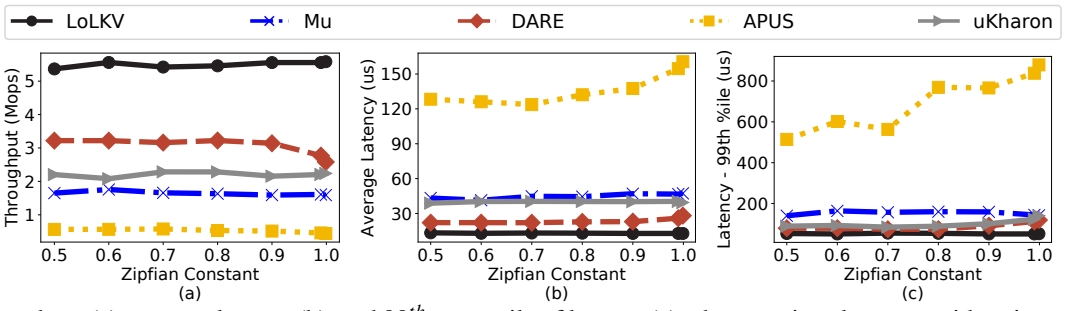

Figure 7: Throughput (a), average latency (b), and 99^{th} percentile of latency (c) when varying skewness with write-only workload.

be deployed with larger number of threads/shards than APUS, Mu, and uKharon. Finally, DARE's batching mechanism batches similar operations only (i.e., a batch can have only `get` or `put` requests). This approach leads to small batches and degrades DARE's performance significantly when the workload has both `get` or `put` requests.

6.2 Latency Evaluation

Figure 6.a shows the CDF of `put` latency for a uniform write-only workload. Other workloads in the YCSB benchmark have a similar pattern. LoLKV lowers the 90th percentile of `put` latency by *30%*, *42%*, *80%* and *91%* compared to DARE, uKharon, Mu, and APUS, respectively.

Figure 6.b and Figure 6.c show the breakdown of the `put` latency into commit time and apply time under the uniform write-only workload. We measure the commit time as the time from when the leader polls a client request until the leader commits the request. LoLKV lowers the 90th percentile of commit latency by *33%*, *73%* and *89%* compared to uKharon, Mu, and APUS, respectively. DARE's commit latency is comparable to LoLKV. However, DARE has a large queuing delay as it polls requests one by one. The impact of this polling mechanism appears in Figure 6.a. LoLKV outperforms other systems as requests are replicated concurrently and committing an operation requires only one RDMA Write operation.

Figure 6.c shows the CDF of the apply time of different systems under the uniform write-only workload. The apply time is the time needed by the system to update the key-value store. LoLKV lowers the 90th percentile of the apply latency by *26%*, *63%*, *39%*, and *45%*, compared to DARE, uKharon, Mu and APUS respectively. In LoLKV, applying an operation requires only updating the hash table to point to the new object. In all other systems, in addition to updating the hash table, the object is copied from the log to the segment store.

6.3 Workload Skewness

We evaluate the impact of workload skewness on the throughput (Figure 7.a), the average latency (Figure 7.b), and the 99th percentile latency (Figure 7.c) when varying the skewness factor between 0.5 and 0.999 for the write-only workload. Results show that LoLKV handles extremely skewed workload efficiently; its throughput and latency are not affected by the workload skewness. With a skewness factor of 0.999, the throughput of DARE, uKharon, Mu, and APUS is reduced by 14%,16%, 6%, and 18%, and their average latencies is increased by 27%, 13%, 4%, and 27% compared to their throughput and latencies with a skewness factor of 0.5.

Increasing the skewness factor does not result in a significant performance degradation for all systems. The reason for this is that with a skewness factor of 0.999, 50% of the requests target only 209 keys. However, we found that these keys are distributed among different shards. Hence, the load generated by the clients is still distributed among all shards. As a result, we do not see a noticeable difference in the performance when comparing uniform and skewed workloads.

Modern key-value stores allow clients to select a partition function that maps keys to shards. These functions often group related keys in a single shards. To better understand the impact of workload skewness on the performance, we conducted an experiment in which we control the percentage of requests targeting a shard. Figure 8.a shows the systems throughput, Figure 8.b shows the average latency, and Figure 8.c shows the 99th percentile latency when we vary the percentage of requests that are handled by a specific shard. For ease of com-

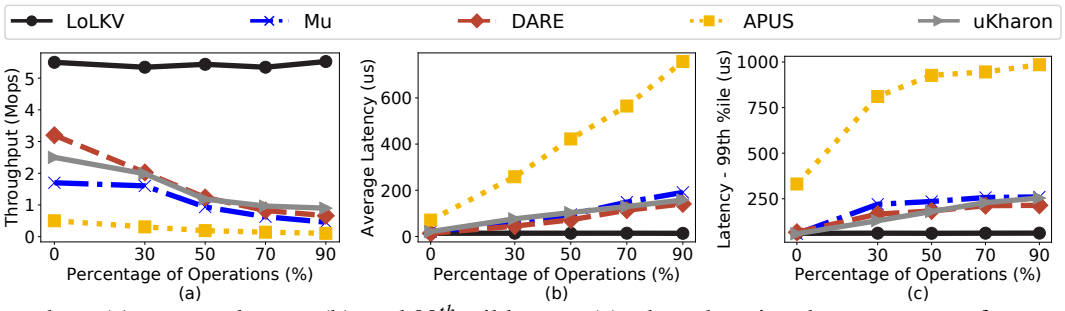

Figure 8: Throughput (a), average latency (b), and 99^{th} tail latency (c) when changing the percentage of requests targeting a shard under write-only workload.

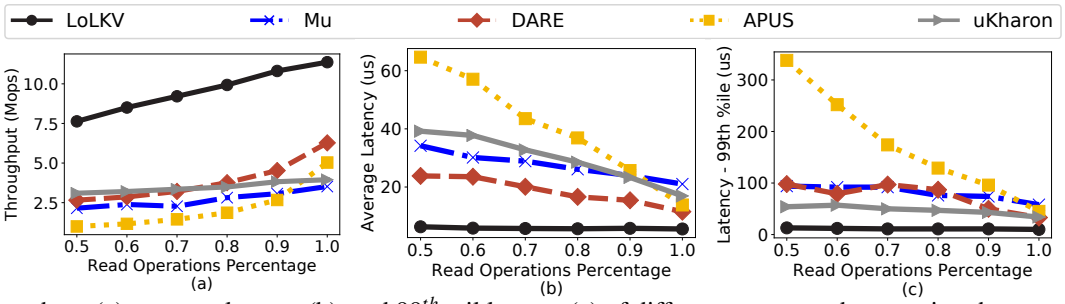

Figure 9: Throughput (a), average latency (b), and 99^{th} tail latency (c) of different systems when varying the percentage of read requests.

parison, Figure 8 also shows the performance when requests are uniformly distributed among shards ("U" in Figure 8). Results show a significant degradation in the performance of DARE, Mu, uKharon, and APUS. For instance, comparing the uniform distribution to when 50% of requests are served by one shard, the throughput of DARE, uKharon, Mu, and APUS is reduced by 61%, 52%, 45%, and 62%, respectively. Similarly, the average latencies of DARE, uKharon, Mu, and APUS are 5.3× , 3.4×, 4.5×, and 5.9× their average latencies with uniform distribution, and their 99th percentile latencies are 2.7×, 3.2×, 4×, and 2.8× of their 99th percentile latencies with uniform request distribution.

LoLKV's performance is not affected by the workload skewness because LoLKV does not shard the key space among worker threads and all threads can participate in serving requests for popular objects. Concurrent `put` operations on the same key can be committed in parallel by different threads. Concurrent updates are ordered when updating the hash table. For all other systems, the commit and apply operations are serialized when there are concurrent updates to the same key and all operations for a popular key are handled by a single shard while other shards could be idle.

6.4 Read-to-Write Ratio

Figure 9 shows the throughput, the average latency, and the 99th percentile latency when varying the `get` requests percentage. Increasing the percentage of `get` requests increases the throughput and lowers the latency of all systems as `get` requests are served by the leader in all systems. Nonetheless, LoLKV outperforms other systems for all read-to-write ratios. For instance, when 70% of requests are `get` requests, LoLKV achieves *2.8*×, *2.75*×, *4*× and *6.3*× higher throughput compared to DARE, uKharon, Mu, and APUS, respectively. LoLKV outperforms Mu and uKharon for read-only workloads because LoLKV uses higher number of threads to serve requests. DARE polls one request at a time which limits its throughput even under read-only workloads.

6.5 LoLKV Failover

In Figure 10, we evaluate the recovery time of LoLKV when the leader fails, including the time it takes to elect a new leader and perform data consolidation. In this experiment, we use the same uniform write-only workload we use in Figure 4.a with the configuration that achieves the maximum throughput of 5.7 Mops. We kill the leader process at the 50 ms mark. The throughput drops to zero when the leader fails and the system remains unavailable until a new leader is elected.

The total recovery time is around *4.5ms*. Detecting the leader failure and electing a new leader takes around *1.4ms*. This is dominated by waiting for 3 heartbeat periods (300μs each) to start leader election. Around *3ms* are spent bringing the new leader up-to-date. Once the new leader is active, the throughput of the system returns to 5.7 Mops. We note that the current implementation of LoLKV is not optimized for fast failure recovery. For instance, in the current implementation one thread sequentially performs data consolidation for all thread segments before the leader can begin serving client requests. This can be parallelized by making each thread update the segments it owns. We will explore optimizing failure recovery in future work.

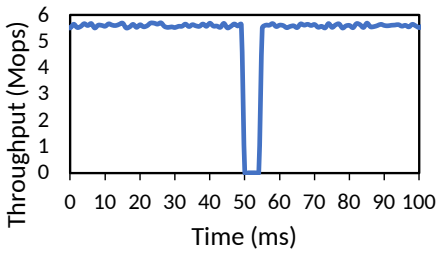

Figure 10: Recovery time of LoLKV.

6.6 Scalability

In this section, we evaluate the scalability as we increase the number of shards or threads. We deploy the systems on three nodes and uniformly distribute shard leaders across nodes, i.e., each node is a leader for one third of the shards and a follower for others. Figure 11 shows the throughput of different systems with increasing number of shards/threads. The results show that LoLKV can scale to efficiently use all hardware threads (i.e., 24 threads) on the three machines and achieve up to 18 Mops which is 4× higher than DARE, 7× higher than uKharon, 10× higher than Mu, and over 36× higher than APUS. LoLKV's performance drops when having more than 24 threads due to fully utilizing all CPU cores.

LoLKV efficiently use all CPU cores because followers in LoLKV are not active during normal operations. Consequently, all available CPU cores on each node are used to host leader shards serving client requests. The other systems do not scale as well because each node is a leader for n shards and a follower for $2\times n$ shards. Thus, two thirds of a node's resources are used by follower processes. Furthermore, in Mu, uKharon and APUS not all threads are used for processing client requests. For each shard process, Mu uses 4 threads on the leader and 4 on each follower, out of which only one of the leader's threads processes client requests. APUS and uKharon use 2 threads on the leader and every follower per shard. DARE is a single-threaded system and thus it scales to a larger number of shards. Each DARE shard uses one thread on the leader and each follower.

7 Additional Related Work

Consensus optimizations. Numerous software-based approaches towards optimizing consensus protocols exist in current literature. EPaxos [24] is a leaderless replication protocol designed to optimize commit latencies. CURP [25] utilizes commutativity to improve replication speeds. While these optimizations improve performance, they still result in latencies in the hundreds of microseconds or millisecond ranges.

Network-accelerated consensus. FLAIR [26], NetChain [27] and NetPaxos [28] utilize programmable switches to accelerate consensus. However, FLAIR only optimizes read operations while NetChain and NetPaxos are only suitable for systems with data sizes of a few megabytes or less.

Micro-scale optimizations. Several software optimizations have been developed to support microsecond range workloads, such as low-latency network stacks [29, 30] and intra-node schedulers [31, 32]. These optimizations are orthogonal to LoLKV, as we explore a novel design for micro-scale linearizable key-value storage.

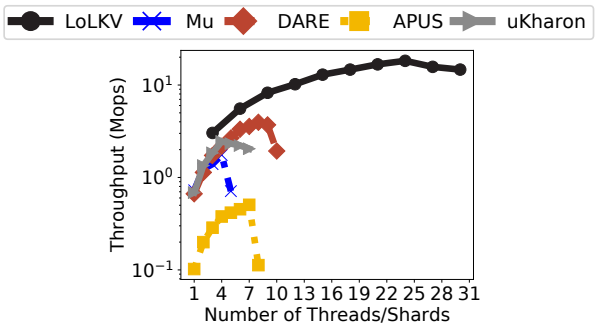

Figure 11: The throughput of different systems when varying the number of shards/threads with leaders distributed on three nodes.

Unreplicated RDMA KV Stores. The clients in Pilaf [13] and XStore [14] use RDMA to access server-side data structures to reduce latency. HERD [12] uses RDMA writes when clients submit requests to the server. The server polls these requests for processing. To leverage multi-core machines, scaling these systems requires key-space sharding and processing all requests for a shard by a thread/process.

Distributed Transactional Systems. DrTM [33] and DrTM+R [34] use RDMA and hardware transactional memory to support distributed transactions. RDMA-based CAS operations are used to replicate local transactions supported by HTM. These works are orthogonal to ours as we focus on using RDMA to build a logless replicated key-value store.

8 Conclusion

We present LoLKV, a novel logless key-value storage system which provides data replication and linearizability guarantees while avoiding the shortcomings of modern log-based key-value stores. LoLKV uses multi-threading to scale and handle client requests, applies operations directly to the underlying data structures of the object store, and uses RDMA to accelerate its operations. LoLKV followers are passive, enabling the efficient use of their resources. Our evaluation shows that LoLKV achieves significantly lower tail latency and higher throughput compared to the state-of-the-art systems. LoLKV shows that strong consistency guarantees can be achieved without being restricted to classical log-based consensus designs.

Acknowledgments

We thank Alex Kogan, Mina Tahmasbi Arashloo, and the anonymous reviewers for their insightful feedback. This research was supported by grants from Oracle and NSERC. Ahmed Alquraan is supported by an IBM PhD fellowship.

References

[1] Xinhui Tian, Rui Han, Lei Wang, Gang Lu, and Jianfeng Zhan. Latency critical big data computing in finance. *The Journal of Finance and Data Science*, 1(1):33–41, 2015.

[2] Stephen F. Elston and Melinda J. Wilson. Big data and smart trading. https://www.risktech-forum.com/media/download/61681/download.

[3] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. Putting the "micro" back in microservice. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 645–650, Boston, MA, July 2018. USENIX Association.

[4] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, 2013.

[5] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. Apus: Fast and scalable paxos on rdma. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 94–107, New York, NY, USA, 2017. Association for Computing Machinery.

[6] Marius Poke and Torsten Hoefler. Dare: High-performance state machine replication on rdma networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, page 107–118, New York, NY, USA, 2015. Association for Computing Machinery.

[7] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond consensus for microsecond applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 599–616. USENIX Association, November 2020.

[8] Rachid Guerraoui, Antoine Murat, Javier Picorel, Athanasios Xygkis, Huabing Yan, and Pengfei Zuo. uKharon: A membership service for microsecond applications. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 101–120, Carlsbad, CA, July 2022. USENIX Association.

[9] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX annual technical conference (USENIX ATC 14)*, pages 305–319, 2014.

[10] Leslie Lamport. Paxos made simple. 2001.

[11] Diego Didona and Willy Zwaenepoel. Size-aware sharding for improving tail latencies in in-memory key-value stores. In *NSDI*, pages 79–94, 2019.

[12] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using rdma efficiently for key-value services. *SIGCOMM Comput. Commun. Rev.*, 44(4):295–306, aug 2014.

[13] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-Sided RDMA reads to build a fast, CPU-Efficient Key-Value store. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 103–114, San Jose, CA, June 2013. USENIX Association.

[14] Xingda Wei, Rong Chen, and Haibo Chen. Fast RDMA-based ordered Key-Value store using remote learned cache. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 117–135. USENIX Association, November 2020.

[15] Rajkumar Buyya, Toni Cortes, and Hai Jin. An introduction to the infiniband architecture. In *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, pages 616–632, 2002.

[16] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 437–450, Denver, CO, June 2016. USENIX Association.

[17] Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Computing Surveys (CSUR)*, 47(3):1–36, 2015.

[18] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. *SIGPLAN Not.*, 35(11):117–128, nov 2000.

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

[20] Diego Ongaro. LogCabin. *https://github.com/logcabin/logcabin*.

[21] Leslie Lamport. The TLA+ Home Page. *https://lamport.azurewebsites.net/tla/tla.html*.

[22] 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. The Design and Operation of Cloudlab. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, pages 1–14, 2019.

[23] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 143–154, 2010.

[24] Iulian Moraru, David G Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 358–372, 2013.

[25] Seo Jin Park and John K Ousterhout. Exploiting commutativity for practical fast replication. In *NSDI'19*, pages 47–64, 2019.

[26] Hatem Takruri, Ibrahim Kettaneh, Ahmed Alquraan, and Samer Al-Kiswany. Flair: Accelerating reads with consistency-aware network routing. In *NSDI'20*, pages 723–737, 2020.

[27] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 35–49, 2018.

[28] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. Netpaxos: Consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, pages 1–7, 2015.

[29] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 325–341, 2017.

[30] Dominik Scholz. A look at intel's dataplane development kit. https://api.semanticscholar.org/CorpusID:11483651, 2014.

[31] Sarah McClure, Amy Ousterhout, Scott Shenker, and Sylvia Ratnasamy. Efficient scheduling policies for Microsecond-Scale tasks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1–18, Renton, WA, April 2022. USENIX Association.

[32] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for usecond-scale tail latency. pages 345–360, 2019.

[33] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 87–104, New York, NY, USA, 2015. Association for Computing Machinery.

[34] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using rdma and htm. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.