FLEET: High-Performance Durable Replicated State Machines using Scattered and Coordinated Log Entries

Hua Fan
Alibaba Cloud Computing
guanming.fh@alibaba-inc.com

Hao Tan
University of Waterloo
hao.tan@uwaterloo.ca

Wenchao Zhou
Alibaba Cloud Computing
zwc231487@alibaba-inc.com

Feifei Li
Alibaba Cloud Computing
lifeifei@alibaba-inc.com

ABSTRACT

Distributed coordination services are fundamental components of distributed systems, employing durable replicated state machines (RSMs) to ensure consistency across replicas and prevent data loss, even in the event of all nodes failing. These services typically rely on persistent logs for rapid recovery, as a universally agreed-upon log allows replicas to restore their state by sequentially replaying ordered log entries. However, the requirement for a *totally ordered* log inherently limits opportunities for parallelism.

This paper introduces Fleet, a high-performance durable RSM protocol that combines a hybrid scattered-entry log with an asynchronous ordered log. Our approach integrates synchronous persistence of scattered entries with asynchronous persistence of ordered entries, ensuring both rapid recovery and high levels of parallelism. Additionally, we propose a parallel applying optimization for the etcd database, named *pre-apply*. Experimental results demonstrate that Fleet significantly outperforms Raft and Scalog in terms of throughput and latency, achieving up to 10× the throughput under specific configurations and scaling effectively across multiple nodes. Additionally, with the pre-apply optimization, Fleet delivers a 10-fold increase in throughput compared to sequential applying on etcd. Although Fleet incurs a 5% overhead in recovery time during leader failure, this delay is tolerable given the rarity of such events.

PVLDB Reference Format:
Hua Fan, Hao Tan, Wenchao Zhou, and Feifei Li. FLEET: High-Performance Durable Replicated State Machines using Scattered and Coordinated Log Entries. PVLDB, 18(5): 1522 - 1535, 2025.
doi:10.14778/3718057.3718077

1 INTRODUCTION

Distributed coordination services, such as etcd [20], Zookeeper [26], and Chubby [9], are essential components in distributed computing. They support critical tasks such as leader election, service discovery, resource management, group membership, and consistent data replication [1], forming the backbone of modern cloud computing infrastructures (e.g., Kubernetes). These services are typically implemented as durable Replicated State Machines (RSMs) [51], employing consensus protocols to ensure state synchronization across nodes. In addition to consistency, several other capabilities are also critical for the effective operation of coordination services:

1. *Durability*: Ensures that the state machine remains consistent and persistent across all nodes, even in scenarios where all nodes crash (e.g., due to disasters or operator errors).

2. *Low Latency*: These systems often function as the control plane for large-scale systems, necessitating rapid response to client requests. Immediate state persistence and swift recovery following restarts are therefore essential.

While RSMs have garnered significant research attention [34, 35, 38, 41, 43, 55, 56], the techniques required to achieve durable RSMs that meet the aforementioned requirements have been less extensively discussed. Current RSM systems for ensuring durability can be categorized into three main types:

(1) **Fail-Stop Assumptions**: Many RSM studies, especially those based on Paxos, assume fail-stop failures, so they have not been designed for rapid recovery. Some approaches never persist data to disk (e.g., CMP [56]), or use ambiguous descriptions of persistence, making it difficult to assess their impact on performance. Protocols like MultiPaxos [55], EPaxos [43], and Mencius [42] simply assume that the entire state on the replicas is made durable by "logging every state change before acting upon or replying to any message" [43]. However, in practice, the state changes can be large or frequent, making recovery costs impractically high.
(2) **Eventual Durability**: Some studies, for better performance, accept the risk of data loss and only guarantee eventual durability [31]. For instance, approaches that use asynchronous storage, like FasterLog [10, 40], are implemented to boost performance. These methods often do not suit our targeted scenarios well.
(3) **Fail-Recovery Model**: Raft uses the fail-recovery model, explicitly incorporating a globally ordered, durable replicated log. This approach makes the protocol easier to understand and implement, leading to widespread adoption in industry [2].

In conclusion, the market favors a synchronized, ordered durability approach for immediate persistence and fast recovery.

However, maintaining the aforementioned logs across replicas introduces significant performance challenges. Extensive prior research has highlighted the poor performance of coordination services such as Zookeeper and etcd [7, 14, 29, 47, 48, 57]. A common misconception attributes this subpar performance primarily to slow durable storage devices. However, despite substantial advances in fast persistent storage technologies, such as NVMe and RAID, the performance issues persist [7]. This "slowness" inherently originates from the consensus protocols, specifically from the waiting time required by the coordination mechanisms needed to maintain order. For instance, the Raft protocol's AppendEntries RPC(see [45], Figure 2) mandates that followers receive and persist entries in accordance with the Log Sequence Number (LSN) order. Furthermore,

Proceedings of the VLDB Endowment, Vol. 18, No. 5 ISSN 2150-8097.
doi:10.14778/3718057.3718077

concurrency control mechanisms, such as locks, are essential for maintaining consistent order while dispatching entries to different replicas within the same sequence. These factors significantly restrict the parallelism of Raft's log entry processing, an issue that cannot be resolved by faster storage solutions. Another perspective is batch processing, which can amortize coordination costs and potentially enhance throughput(e.g., Scalog's ability to maintain both ordering and parallelism, detailed in §2.3). However, the resultant high latency of batching may be prohibitive for many applications, especially in light of the low-latency requirements.

Our key observation is that current solutions unnecessarily sacrifice parallelism to maintain a totally ordered durable log. It is well-known that achieving both serial and concurrent persistence within a single log is infeasible. However, a hybrid approach can achieve the advantages of both methods with minimal cost: scattered log entries for fault tolerance and eventually durable ordered logs for fast recovery. Since these two forms are complementary, they do not require doubling the disk space for the entire content.

Based on our observations, we introduce Fleet, a scalable RSM that merges a scattered-entry log with an asynchronous ordered log. *First*, Fleet utilizes a stable leader to efficiently determine the order of log entries, leveraging a fleet of storage nodes as an *unordered entry set* for synchronous persistence. Significantly, we discovered that removing the *ordering restriction* during the transmission and persistence phases of log entries markedly increases system concurrency. Furthermore, these scattered log entries endow Fleet with enhanced scale-out capabilities. *Second*, Fleet integrates the *synchronous* persistence of scattered entries with the *asynchronous* persistence of ordered entries. This hybrid methodology facilitates rapid recovery while maintaining high levels of parallelism. *Third*, we identify that etcd's sequential applying process can become a performance bottleneck in highly parallel systems like Fleet, thus propose a parallel applying technique for the etcd database, termed *pre-apply*, which requires the co-design of the consensus layer and the database layer. The concept behind pre-apply is akin to parallel replay [22, 23] in primary-backup systems with log shipping. However, unlike traditional methods that demand a dependency graph (among transactions or rows) generated by the primary, pre-apply eliminates the need for such dependencies. Our optimization involves maintaining an additional visibility watermark, below which monotonic prefix applying has been completed. This approach facilitates efficient parallel applying without the requirement for dependency graphs.

To demonstrate the performance of Fleet, we implemented the protocol based on the open-source etcd [20] version 3.5, and evaluated it with two different upper-layer applications: an in-memory key-value store (MemKV) and the fully-featured etcd, which utilizes a persistent B-tree-based multi-version database (MVDB). Our experimental results indicate that Fleet significantly outperforms both Raft and Scalog in terms of throughput, without sacrificing latency. Specifically, in the MemKV application, Fleet achieves up to 7× the throughput on the same hardware configuration, and up to 10× the throughput when utilizing additional local disks. In scalability tests using up to 15 machines, Fleet achieved 5 million operations per second (Mops) for writing small values (64 bytes), fully saturating the thread responsible for applying commands. For larger values (4 kilobytes), Fleet effectively utilized the 20 Gbps network cards on our server machines, approaching the performance upper bound of a non-replicated state machine. As a trade-off, Fleet incurs a modest overhead of 5% in recovery time, translating to an acceptable 50 ms delay upon leader failure, an event which is expected to be infrequent. To assess the efficiency of our pre-apply optimization, we conducted experiments on a 5-node etcd cluster. The pre-apply optimization increased throughput by a factor of 4.4 while consistently maintaining lower latency compared to sequential applying. In a scale-out cluster configuration, the pre-apply enabled etcd achieved around 600k operations per second, which is 10× faster than using sequential apply.

In summary, this paper makes the following contributions:

- We introduce Fleet, a scalable and durable state machine replication protocol that significantly enhances parallelism and scalability through the use of scattered log entries.
- We design a novel recovery mechanism combining synchronous scattered persistence for high performance with asynchronous centralized persistence for fast recovery.
- We propose a parallel RSM application optimization for etcd, utilizing MV storage and the parallel capabilities of Fleet.
- We implement and evaluate Fleet using a well-established open-source coordination service. Our experimental results demonstrate that Fleet significantly outperforms both Raft and Scalog in terms of throughput and latency.

2 MOTIVATION AND BACKGROUND

This section starts with target scenarios and essential characteristics of durable RSMs for use cases. Then, background on durable RSMs is given, with Raft as an example. Finally, it identifies limitations of current distributed log solutions in coordination services.

2.1 Motivation

Orchestration systems such as Kubernetes (K8S) [3], Twine [53], and Autopilot [27] typically utilize event-based declarative engines and loosely coupled distributed services. A key component of these systems is a strongly consistent and persistent data store, such as etcd [20], ZooKeeper [26], or Chubby [9]. The performance of etcd significantly impacts the scalability and user experience of K8S [29, 49, 57]. As the control plane of K8S, etcd is critical for ensuring predictable **low latency** for persistence, even in the presence of **stragglers** and during **recovery**. For instance, in the event of service anomalies, it is essential to promptly report node status and rapidly reroute user traffic to healthy nodes.

As K8S is increasingly adopted in cloud environments, and as applications on K8S become more reliant on etcd, the performance issues become more pressing. This calls for a **high-throughput** solution that can scale out effectively. Sharding is a commonly employed technique to enhance the scalability of an RSM. It divides the state space into disjoint partitions, each managed by an independent RSM instance. While this method improves parallelism, it sacrifices the guarantee of total order for operations involving multiple RSM instances. Moreover, sharding is susceptible to **hotspot workloads**. For example, when a small number of applications on K8S generate spiky workloads (e.g., a sudden influx of new jobs or a failure impacting numerous nodes), the corresponding etcd shard

Table 1: Fleet and existing systems compared against performance requirements

	Fleet	Raft	CMP	Corfu	Scalog	EPaxos
Low Latency	✓	✓	✓	✓	✗	✓
High Throughput	✓	✗	✓	✗	✓	▵
Fast Recovery	✓	✓	✗	✓	✓	✗
Straggler-friendly	✓	✓	✓	✗	✓	✓
Hotspot Tolerance	✓	✗	✓	✗	✓	✗

may become overloaded, as other shards cannot assist in offloading the workload of the hotspot partition.

As summarized in Table 1, no existing systems fulfill all the highlighted requirements. Raft [45] persists log entries in order, thereby limiting its throughput and scalability. Compartmentalized MultiPaxos (CMP) [56] does not maintain a persisted ordered log, resulting in high recovery overhead. Distributed log protocols such as Corfu [4] and Scalog [15] enhance concurrency by utilizing additional servers. However, Corfu's limited throughput motivated the development of Scalog [15]. As outlined in § 2.3, Corfu is vulnerable to stragglers and negatively impacted by single-node hotspots and Scalog trades off latency for throughput. EPaxos [43] uses topological sorting for ordering but suffers under high contention (e.g., accessing the same key in a key-value store). As our aim is a general-purpose RSM where all concurrent operations may conflict, EPaxos is likely to underperform in such scenarios.

2.2 Durable RSM

Replicated State Machines (RSMs). Consensus protocols are vital to RSMs, constructing a continuous log of client *commands* and coordinating nodes to ensure that all replicas maintain identical log entries in the same order, even amid failures. Each replica's state machine *executes* commands deterministically by *applying* the log entries in sequence and returning results to clients. Consensus protocols typically guarantee two fundamental properties when a majority of replicas are operational: (1) *safety*, which ensures consistency among replicas at all times, and (2) *liveness*, ensuring a client command will eventually be appended to the log.

Durability. Durability guarantees that no operation acknowledged to clients is lost, even if all nodes fail. This is possible because each node can recover its state from data stored on disk after a crash. However, the recovery speed critically impacts the actual availability of the coordination service. Consequently, selecting a durable RSM that supports rapid recovery is a careful decision. For instance, while both unordered and ordered durable log entries ensure durability, their recovery costs vary significantly. Ordered logs can be loaded sequentially with ease, whereas unordered logs must be loaded into memory and sorted before applying the entries. For large data volumes (e.g., several hours of logs), the sorting process can exceed available memory. Some scale-out optimizations (e.g., CMP) further complicate recovery by requiring log entries to be gathered from remote machines before sorting, incurring significant overhead. Thus, widely-used open-source systems in the industry, such as etcd and ZooKeeper, employ ordered persistent logs.

Raft. Raft is a widely implemented consensus protocol, especially in industry. It employs a *leader election* process to appoint a stable leader for each *term* from a group of nodes. The leader decides the order of client commands. Raft achieves fault tolerance through *log replication*. Specifically, the leader assigns a unique LSN to each log entry and determines when it is safe to apply that entry. The leader commits an entry only after replicating all preceding entries to a majority of nodes. Subsequently, followers apply the entry after the leader has committed it. Each replica maintains a local persistent log where log entries are written sequentially.

To ensure global consistency, Raft introduces several coordination points during log persistence: followers must wait for commit decisions from the leader, leaders must wait for acknowledgments from a quorum of followers before committing, and each follower must persist log entries in LSN order, waiting for all preceding entries to be persisted first.

2.3 Distributed Logs

Distributed replicated logs [4, 5, 15] are often engineered with scalability as a primary objective. Unfortunately, these designs do not fulfill the essential functional requirements for coordination services, as discussed in Section 2.1.

Sharding is a commonly employed technique to enhance concurrency. To maintain the order of log entries across different shards, all shards must agree on the mapping of each LSN to the storage location of its corresponding entry.

Corfu [4] utilizes a static round-robin strategy for mapping entries to shards and their locations. However, this placement strategy lacks the flexibility to avoid imbalanced load distribution and the occurrence of straggler shards. Consequently, some shards can become overloaded while others remain under-utilized, negatively impacting the parallelism of log persistence.

Scalog adopts a persistence-first and assign-LSN-later model: (1) Entries can be written to any shard without the need for LSNs. (2) However, to assign LSNs to persistent log entries within an epoch, Scalog must achieve consensus (using Paxos) on the LSN mapping across all shards before committing the log entries. Scalog commits a command only after consensus is reached across all shards for the LSN assignment of all commands within an epoch.

Scalog maintains a total ordering among entries while facilitating concurrency across shards. However, this design inevitably introduce additional latency in two key areas: (a) The necessity for an additional write and coordination step (first, for entries without LSNs, and second, for mapping the LSNs to the respective entries); (b) Scalog's epoch-based batching mechanism. As a result, this design leads to increased latency, representing a "fundamental limitation of Scalog" [15].

Moreover, both Corfu and Scalog require that replicas within each shard be identical, meaning all log entries across shard-wide replicas must maintain the same order. The coordination needed to ensure this uniform order incurs substantial overhead.

3 FLEET ARCHITECTURE

This section overviews Fleet's architecture, introduces its modules, shows an example of handling client requests, and discusses two deployment types: co-located and separated modes.

3.1 Modules

Figure 1 depicts the architecture of Fleet, which consists of Replica Nodes (RNodes) and Storage Nodes (SNodes). Note that RNode and

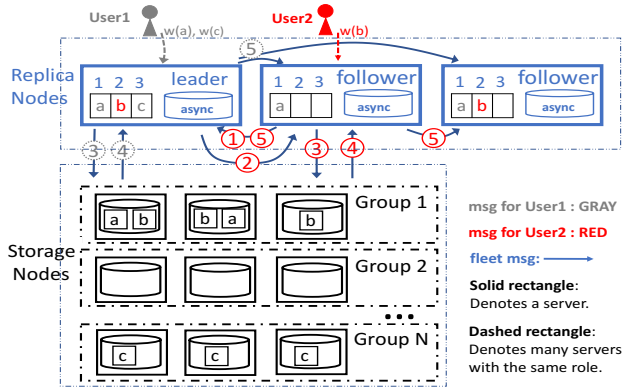


Figure 1: Architecture of Fleet. RNodes execute client requests; Snodes persist scattered and unordered entries.

SNode are logical units; in practice, they may be colocated on the same physical server (§ 3.3).

3.1.1 RNodes. An RNode maintains the *state* of the RSM along with a consensus-agreed *log* array. This design is similar to other consensus protocols such as Raft and Zab. However, unlike these protocols, the log entries in our system are not subjected to a voting process by the group of RNodes. Additionally, unlike Raft, which ensures durability by storing the log on stable storage devices (e.g., disks), our *log* resides in volatile memory.

A notable innovation introduced by Fleet is the **Asynchronous Persistence (AP) module**, which persistently writes the *log* to disk in an asynchronous manner. This persisted version of the log, referred to as the *Ordered Log* (Olog), may occasionally miss some of the tail entries if the log array undergoes continuous updates. Importantly, the AP module is not involved in the critical path of Fleet's normal operations.

Fleet can still operate without the AP module, albeit with slower recovery performance, as it does not lose data and can be fully recovered from the storage layer alone. The AP module is introduced to expedite recovery by reducing the time required to collect log entries from storage nodes. The AP module ensures the presence of $f+1$ local Olog files to tolerate f failures, or alternatively, it can utilize a remote file, provided that it is ordered and replicated.

RNodes process client requests. When a client request is received, Fleet employs its consensus mechanism to achieve log replication across RNodes and subsequently applies the operation to the system state (§ 4.2). One of the RNodes is elected as the leader(§ 4.3), while the remaining RNodes serve for fault tolerance.

3.1.2 SNodes. SNodes store scattered log entries used for recovery. There is no ordering guarantee for entries that are durable on SNodes; hence, each SNode is expected to have multiple disks to enhance write concurrency. Each SNode provides simple interfaces for storing and retrieving durable data:

- `SaveRPC(`*entry*`)` persists a given entry to disk.
- `Since(`*lsn*`)` retrieves entries with LSN greater than *lsn*.
- `Trim(`*lsn*`)` removes entries with LSN smaller than *lsn*.

For clarity of presentation, we organize the storage nodes into groups, each comprising $2f+1$ storage nodes. However, the correctness of Fleet does not depend on statically binding each storage node to a specific group. This approach contrasts with protocols such as CMP [56] or *grid quorum* [24], which require nodes to form predefined row or column layouts. Fleet only necessitates that any group possesses enough replicas to effectively tolerate faults.

Moreover, Fleet does not mandate the presence of multiple groups. In fact, our experiments (§6.2) indicate that even with a single group (co-located deployment in §3.3), writing scattered log entries is significantly faster than writing a totally ordered log.

3.2 Fleet Example

Figure 1 illustrates a scenario with three concurrent operations: $w(a)$, $w(b)$, and $w(c)$. The operation $w(b)$, served by a non-leader RNode, has its message flow indicated by red circles. Within the Fleet protocol, this RNode *proposes* appending the byte array b to the log. ① and ② involve obtaining an ordering LSN from the leader. ③ and ④ show the log entry persisted by an arbitrary set of $f+1$ nodes within a storage group to tolerate stragglers. ⑤ depicts the replication of the persisted log entry to other replicas. Each RNodes *applies* the committed entry according to the LSN order. Therefore, the entry can be applied by the receiving RNode as long as its preceding entry has been applied. Concurrently, the AP modules in the RNodes asynchronously write the ordered log to durable storage for rapid recovery.

The message flow for operations $w(a)$ and $w(c)$ follows a similar pattern but saves a round-trip for obtaining the LSN, as they are directly served by the leader. These operations are persisted in parallel, ensuring high concurrency without coordination.

Summary. This example demonstrates several advantages: 1. Unlike Raft, where log entry persistence must be ordered, our system allows out-of-order persistence (e.g., a and b can be persisted in different orders on different machines), thus increasing concurrency. Additionally, persisting entries on Snodes enhances scalability (e.g., c is persisted in group N on SNodes instead of group 1 or RNodes). 2. Compared to consensus protocols lacking a durable Ordered Log (e.g., CMP), our system's AP module facilitates faster recovery.

3.3 Deployment

Figure 2 illustrates the two deployment modes of Fleet.

Co-located Mode: In this deployment, an RNode and an SNode are integrated into the same server, forming a single group of SNodes. While this setup is similar to the conventional Raft deployment, there are significant differences in the persistence mechanism of log entries. In Raft, log entries are written to disk in a strictly ordered manner, facilitating streamlined recovery through sequential reads from the disk. Conversely, in Fleet, log entries are dispersed across the disk without maintaining a strict order, which enhances concurrency. Nevertheless, without an AP module, recovery becomes more complex, requiring the aggregation and reordering of all log entries to reestablish the correct sequence. Under high load conditions, the volume of data needing sorting may surpass available memory capacity, leading to prohibitively slow recovery times and rendering this design impractical.

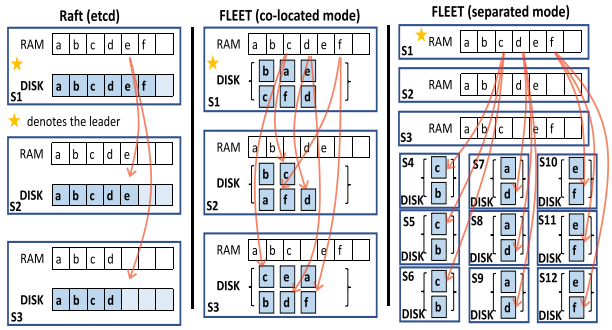


Figure 2: Illustration of log entry layouts: (1) Raft: Entries must be persisted and transmitted in order. Recovery: Reading the ordered entries from disk. (2) Colocated Fleet: Entries may be out-of-order. Naïve Recovery: Requires sorting unordered entries from disk. (3) Separated Fleet : Entries can be stored across servers. Naïve Recovery: Requires collecting and sorting entries from all servers to find committed entries. This figure emphasizes the advantages of using scattered log entries, hence it does not include the AP model.

Separated Mode: In this deployment configuration, RNodes and SNodes are deployed on separate servers. To enhance scalability, the system can be configured with multiple SNode groups. Each SNode group employs a majority quorum mechanism to replicate log entries and tolerate up to f failures. Therefore, each group necessitates $2f + 1$ nodes.

Due to space constraints, this paper presumes a static deployment configuration. Nonetheless, we posit that Raft's reconfiguration mechanisms can be adapted to Fleet.

4 PROTOCOL

Given Raft's widespread adoption and understandability in the industry, this section focuses on comparing Fleet with Raft to facilitate a deeper understanding of the protocol. Table 2 summarizes the similarities and differences between Fleet and Raft.

4.1 System Model

Assumptions. The network is asynchronous, meaning messages can be lost, delayed, or transmitted out of order. Servers can fail by stopping, but may later recover from a stable storage state and rejoin the cluster. Byzantine failures are not considered. The safety of our protocol does not depend on clock synchronization. However, skewed clocks may impact availability (the ability to respond to clients in a timely manner).

Alternatively, availability issues can be addressed under the assumption of bounded clock skew (see §5.6 in [45]). On this basis, the leader could utilize a read lease [21] or other optimizations [44] to enhance read performance.

Basic Rules. We adopt the concept of a *term* from Raft, which segments time into disjoint intervals, akin to the concept of a view in Viewstamped Replication (VR) [41]. Our protocol adheres to the basic rules of Raft as outlined below:

Each server maintains a current `term` variable, included in every outbound message. Servers *only process normal protocol messages* with *terms* matching their current *terms*. If the sender's term is outdated, the receiver rejects the message. Conversely, if the sender's term is more recent, the receiver updates its `term` variable and purges all unprocessed messages and incomplete log replication procedures (i.e., `Propose`) with stale terms from its buffers.

Consistent with Raft, when the `term` is updated, a leader from the preceding term must transition to a *follower* role. Additionally, the `term` must be stored on durable storage to ensure persistence across power failures and reboots.

Each term includes an *election period* and a *normal operation period* if a leader is elected. Since each server only processes messages tagged with the matching `term` during normal operation periods, we do not explicitly include the `term` in the RPC parameters in the following pseudocode.

4.2 Log Replication

The algorithms for log replication are detailed in algorithm 1, algorithm 2, and algorithm 3. Here, `R` denotes the set of all replica nodes, while `G` denotes the set of all storage node groups, each comprising $2f+1$ storage nodes. Storage nodes (SNodes) may maintain multiple write-ahead log (WAL) `files` across different disks and employ an in-memory buffer, `cached`, for recently saved entries pending flush to AP. This buffer facilitates recovery operations. Each replica node (RNode) maintains a `state` and a `log` array for applied entries, with `lastApplied` LSN indicating the latest applied slot. The leader node additionally maintains an `assigned` array to track assigned LSNs and their corresponding entries.

The protocol begins with a client sending a command to an RNode, which then invokes the `Propose` procedure. It starts by retrieving an LSN from the leader, which determines the entry's ordering. Subsequently, it selects an arbitrary group for persistence. Upon confirming that the entry is persisted on at least $f + 1$ nodes of the group, the log entry is considered committed and can be replicated to the `log` array of all RNodes. Each RNode may receive persisted log entries out of order, resulting in holes within the `log` array. However, the `commitIndex` serves as a cursor, ensuring that the prefix preceding the `commitIndex` is free of holes, and all committed entries maintain non-decreasing term values (**Rule 1**). Entries between `lastApplied` and `commitIndex` can be sequentially applied to the state (**Rule 2**). Once the RNode applies a command, the client receives a response indicating that the entry has been applied to the RSM (**Rule 3**).

The procedures on the storage nodes, as illustrated in algorithm 3, simply save the entries to an arbitrary WAL file. Storage nodes also temporarily buffer the data in `cached` before it has been confirmed in in APs. This optimization aims to facilitate rapid recovery, which will be elaborated upon in subsection 4.3.

In comparison with Raft, Fleet achieves higher concurrency in two dimensions: (1) Log entries can be independently transmitted and persisted without the need for coordination, whereas Raft requires log entries to be transmitted and persisted in LSN order, limiting concurrency in multiple scenarios. (2) Raft treats all nodes as identical, without a distinction between RNodes and SNodes, effectively resulting in a single SNode group that cannot scale out. Fleet's architecture, however, can enhance scalability by adding more SNode groups. Additionally, Raft couples RNodes and SNodes together, necessitating that the number of RNodes equals

Table 2: Similarities and Differences between the Raft and Fleet Protocols

		Raft	Fleet
Safety		State Machine Safety	
Liveness		Progress When Clock Skew is Bounded*	
Failure Model		Failure Recovery	
Log Replication	Persist Choice	Single Group	(for each log entry) Random Group
	Order on Disk	In-order (Sync)	Scattered Entries(Out-of-order, Sync)
			In-order (Async)
Leader Election	Vote By	Single Group	All (SNode) Groups
	Recovery	Apply Ordered Entries	Apply Ordered + Scattered Entries

* As defined in [45], §5.6: $broadcastTime \ll electionTimeout \ll MTBF$.

the number of SNodes. Fleet, however, decouples these components, allowing for different f values for RNodes and SNodes in certain scenarios to further optimize resource utilization.

Garbage Collection(GC). Scattered log entries can be deleted from disk to reclaim storage space once they can be reliably recovered from the `Olog`. The leader maintains the maximum LSN that has been flushed to the ordered log in all AP modules and uses the `Trim` API to notify SNodes. SNodes can then lazily delete all files that contain only older LSN entries.

Consequently, only a small portion of log entries consume storage capacity in both SNodes and AP modules. During GC, SNodes do not need to rescan the file contents because the WAL files are of fixed length, and each file name contains the maximum and minimum LSNs. The `Trim` procedure can determine which files can be safely deleted solely based on their file names, while other files can be temporarily retained.

Similarly, the trim mechanism can be employed to clear outdated log entries from the cache. Moreover, upon reboot, an SNode only needs to load the small subset of log entries with LSNs greater than the trim LSN into memory.

Checkpointing. Checkpointing is a widely employed optimization in RSMs and databases to facilitate rapid recovery. Although it involves asynchronously dumping the state of an RSM to expedite recovery [41], the approach and granularity differ significantly from those in Fleet.

Checkpointing typically involves asynchronously dumping the entire RSM state or database to avoid the costly replay of the entire log from the beginning. This process is performed infrequently in practice, often at hourly intervals. Checkpointing assumes an totally ordered log that can be directly loaded, whereas Fleet's asynchronous `Olog` dump avoids the overhead of assembling a scattered log into that ordered log. Fleet's asynchronous dumping operates at a sub-realtime granularity, calling flush by sub-second intervals. Fleet's checkpoint generation is akin to that of Raft, with the notable exception that it only compacts the entries that are in the `Olog` for simplicity.

Filling Holes. A "hole" refers to a scenario where a slot has been reserved but not committed to the `log[]`. Such holes may occur during normal operations due to an incomplete `Propose` procedure, often caused by an RNode crash. Unfilled holes pose significant liveness issues for the system.

Holes can be readily identified when the `commitIndex` remains stagnant and is lower than the leader's `index`. Non-leader RNodes can detect and fill these holes by heartbeating with the leader. In contrast, the leader must retrieve the entry e from the `assigned` array and execute lines 13- 15 of algorithm 1, which cover the persist and replicate operations. These actions are idempotent, guaranteeing that even if both the leader and the original RNode execute them, the final state remains consistent, with e correctly written to `log[e.lsn]`.

4.3 Leader Election and Recovery

To ensure that each term has at most one leader, the leader election protocol must achieve consensus among all nodes and persist the election decision (new term and leader ID) across all storage groups before processing any new client requests. Subsequently, each SNode group will refrain from persisting entries from older terms, thereby preventing any former leader from proposing new commands. It is important to highlight that our solution is resilient to stragglers, as each group requires the acknowledgment of a majority quorum.

4.3.1 Leader Election. The leader election protocol in Fleet is fundamentally derived from Raft: nodes maintain their roles through heartbeats and use heartbeat timeouts and election messages to transition between leader, candidate, and follower roles (detailed in Section 5.1 of [45]). However, due to the decoupling of RNodes and SNodes in Fleet, there are notable differences in the election logic: (1) The leader is selected from one of the RNodes; however, decisions are made exclusively by the SNodes (SNodes never become candidates and RNodes never grant votes). (2) Whereas Raft operates with a single SNode group, Fleet comprises multiple SNode groups, necessitating that the final leader coordinate a consistent result across all groups.

Strawman Version: Each RNode R_i requests each SNode group to vote for itself(R_i). Each SNode grants the vote to the first request in a term and rejects subsequent requests, responding with the granted candidate ID. A majority vote from the nodes in a group is required for an RNode to *win* that group. For any given SNode group, R_i can encounter one of three outcomes: (1) it wins the group, (2) another RNode wins the group, or (3) no RNode wins the group. Upon receiving results from all groups, R_i may discover (1) very fortunately, a single RNode has won all groups, or (2) no RNode has won all groups. In the first scenario, the election concludes. In the second scenario, a new term is initiated, and after waiting a randomized interval, the process is retried. The probability of each group independently electing the same RNode is low, prompting the need for an optimization.

Front-Runner Version: A pre-election round is conducted among the RNodes before the strawman version. During this round,

Algorithm 1: Log Replication Procedures on All RNodes

Volatile State:

1 `state`: The state machine.
2 `term`: Current term (see subsection 4.1).
3 `lead`: Leader of this term.
4 `log[]`: RSM execution history.
5 `lastApplied`: Index of the last log entry applied to `state`.
6 `commitIndex`: Highest committed log entry index.

Persistent State:

7 `R`: Set of RNodes.
8 `G`: Set of SNode groups, each group has $2f + 1$ SNodes.
9 `Olog[]`: Asynchronously persists the prefix of `log[]` indexed up to `commitIndex`.
10 **Procedure** `Propose(`*command* c`)`
11 $l \leftarrow$ `lead.SeqRPC(`c`)`
12 $e \leftarrow \langle$lsn: l, cmd: c, term: `term`$\rangle$
13 **repeat** `Persist(`e`)` **until** `Persist(`e`)` *returns True*;
14 **parallel-foreach** $n \in$ `R` **do**
15 n`.ReplicateRPC(`e`)`
`/* Persist entry e on a random group of nodes */`
16 **Procedure** `Persist(`*entry* e`)`
17 *succ* $\leftarrow 0$
18 **parallel-foreach** $n \in g$, `randomly choose` $g \in$ `G` **do**
if n`.SaveRPC(`e`)` *returns success* **then**
19 *succ* ++
`/* Check if the majority has succeeded */`
20 **return** *succ* $\geq f + 1$
21 **Procedure** `ReplicateRPC(`*entry* e`)`
22 `log[`e`.lsn]` $\leftarrow e$

Rules for all RNodes:

23 **R1**: If `log[`$i + 1$`]` (where $i =$ `commitIndex`) is assigned and `log[`$i + 1$`].term` $\geq$ `log[`i`].term`, then increment `commitIndex`.
24 **R2**: If `commitIndex` > `lastApplied`, increment `lastApplied`, apply `log[lastApplied]` to the state.
25 **R3**: If a command is received from a client, call `Propose()` and respond after the entry is applied to the state.

RNodes elect a front-runner (e.g., using Raft). Only the front-runner proceeds to participate in the subsequent strawman version. With this optimization, it is likely that only one candidate will participate in the strawman version, resulting in the front-runner securing leadership within a single round trip time, regardless of the number of groups and nodes. It is important to note that the front-runner must gather votes from all the storage groups; otherwise, certain Snode groups may remain unaware of the new leader's emergence and continue to accept requests from the old leader.

4.3.2 Recovery. Upon election, the new leader must recover all committed entries from the SNodes before serving any new client requests. A naïve approach to recovery involves loading and sorting

Algorithm 2: Log Replication Procedures on the Leader

Volatile State:

1 `assigned[]` : Array to hold entries assigned LSNs.
2 `index`: LSN to be assigned to the next slot; initially set to 1.
3 **Procedure** `SeqRPC(`*entry* e`)`
4 `assigned[index]` $\leftarrow e$
5 **return** `index++`

Algorithm 3: Procedures on All SNodes

Volatile State:

1 `cached`: In-memory cache of log entries.

Persistent State:

2 `files`: A set of WAL files to persist scattered log entries.
3 **Procedure** `SaveRPC(`*entry* e`)`
4 $f \leftarrow$ randomly choose $f \in$ `files`
5 **if** f`.Save(`e`)` **then**
6 `cached[e.lsn]` $\leftarrow e$
7 **return** *success*
8 **return** *fail*
9 **Procedure** `Since(`*lsn* l`)`
10 **return** $\{x \mid x \in$ `cached` $\wedge\ x.lsn > l\}$

all scattered log entries. However, we propose an innovative recovery mechanism in Fleet that combines synchronous persistence with asynchronous ordered logging.

Algorithm 4 illustrates the pseudocode for the recovery process. During recovery, the elected leader executes the `BecomeLeader` procedure. Initially, it replays the `Olog` if it contains a more extensive log than its local `log`. If the leader lacks a local `Olog`, it can read the files from a remote node. Notably, even if the `Olog` is empty due to the AP module's unavailability, the correctness of the recovery remains unaffected.

The leader then resolves the entries absent from the `Olog` by invoking the `Since` RPC on all storage nodes. Typically, the number of these entries is expected to be small, as it depends on the lag between asynchronous logging and synchronous persistence. We collect entries from all groups, requiring responses from a majority of nodes within each group. These retrieved entries are processed by the `Recovery` procedure, which reconstructs the `log` array by resuming each log entry in LSN order.

The process involves sorting the entries by LSN in ascending order and breaking ties by term in descending order. Only one entry per LSN is applied, skipping subsequent entries with an LSN equal to the `applied` LSN (line 15). Recovery stops when no log entry satisfies the following conditions: (1) The entry's LSN is equal to `curLsn` + 1. (2) The entry's term is greater than or equal to the term of the entry in the previous slot.

Lines 23 and 16 of the pseudocode correspond to scenarios where specific conditions are violated. The case in Line 23 is straightforward; however, Line 16 requires further explanation: a) The entry e has the minimum LSN among entries with `lsn` > `curLsn`; b) The above two conditions cannot be simultaneously satisfied for e.

Algorithm 4: Recovery Procedures on the New Leader

```
   Note: commitIndex is automatically refreshed per Rule 1.
1  Procedure BecomeLeader()
2      if len(Olog) > commitIndex then
3          log ← Olog
4      ents ← ∅
       // Collect entries from all SNode Groups
5      parallel-foreach n ∈ ⋃_{g∈G} do
6          ents ← ents ∪n.Since(commitIndex)
7      Recovery(ents)
8      assigned ← log
9      index ← commitIndex + 1
10 Procedure Recovery(entry[] ents)
       // Sort by LSN asc (Prim.), Term desc (Sec.)
11     ents ← ORDER ents BY ⟨LSN ASC, Term DESC⟩
12     curTerm ← log[commitIndex].term
13     curLsn ← commitIndex
14     foreach e ∈ ents do
15         if curLsn = e.lsn then
               continue
16         if e.term < curTerm then
               break // a hole, explained in § 4.3.2
17         if e.term > curTerm then
               curTerm ← e.term
18         if e.lsn = curLsn + 1 then
19             Persist(e)
20             log[e.lsn] ← e
21             curLsn ← e.lsn + 1
22             continue
23         break // LSN curLsn + 1 is a hole
```

Regarding *a)*, since the entries are sorted by LSN in ascending order, curLsn acts as a movable cursor. Entries with an LSN less than curLsn are processed before curLsn is incremented, and Line 15 is not satisfied, hence e.lsn > curLsn.

Regarding *b)*, even if e.lsn = curLsn + 1, Line 16 indicates that e does not meet the condition because its term is too small (it already has the maximum term among entries with the same LSN).

4.4 Correctness

Fleet provides the same liveness and safety guarantees as Raft. This section informally discusses these properties in Fleet.

4.4.1 Liveness. Similar to Raft, Fleet relies on a stable leader to ensure progress. In our implementation, Fleet uses Raft to elect a front-runner, which must be accepted by all SNode groups before being recognized as the new leader. Therefore, Fleet shares the same liveness properties as Raft. The additional SNode voting and persisting processes require a majority quorum within each group, thus posing no additional liveness challenges.

Note: Unlike Raft, which uses $2f+1$ nodes to tolerate f failures, Fleet may increase the number of SNode groups to enhance overall system performance and scalability rather than to increase fault tolerance. Thus, even with $(2f+1)\times(N+1)$ nodes (where N is the number of groups), Fleet's fault tolerance guarantee remains f.

4.4.2 Safety. The correctness of a RSM is defined by state machine safety, which can be stated as: if a server has applied a log entry e_1 to its state machine, no other server will ever apply a different log entry e_2 with $e_1.lsn = e_2.lsn$.

It is clear that within a given term, there is only one leader, and this leader ensures that each slot is assigned a unique entry. Therefore, the key to state machine safety is maintaining correctness during term transitions: every committed operation must survive into all subsequent terms at the same position in the log[]. This condition evidently holds in the initial term. Using induction, we must prove that if an entry e satisfies this condition in term t, it will also satisfy the condition in term $t+1$.

As a committed entry, e must satisfy the following: (1) It is persisted in $f+1$ SNodes within a group. (2) e is the entry with the highest term among entries in Recovery with the same LSN $e.lsn$.

Because e persists across e.term and its recovery ensures that any new leader will assign LSNs greater than $e.lsn$ for new commands, the Recovery procedure guarantees that e will be restored in term $t+1$. Hence, correctness is maintained during term changes.

5 OPTIMIZATION: ETCD PARALLEL APPLY

In the existing etcd system, a consensus layer (i.e., the Raft package) produces a commit log, while a separate DB layer (i.e., the mvcc package [16]) consumes this commit log. Both of these packages are integrated within the etcdServer package [17], enabling seamless communication between packages and across hosts. One can easily replace the Raft package with Fleet in the consensus layer without necessitating modifications to other system components, as both protocol inputs consist solely of proposed byte arrays from clients, while the outputs are sequential log entries. However, an accelerated consensus layer may render the etcd database layer a bottleneck, given that the etcd database processes log entries sequentially.

To address this issue, we propose a co-design of the consensus and application layers to enable parallel application, leveraging the high parallelism offered by Fleet and the multi-versioning capability of etcd. There are two sources of increased parallelism: (1) Instead of assigning the next LSN to any arbitrary entry, the leader assigns the next LSN to the winner of the execution of DB concurrency control. This allows for higher parallelism compared to sequential applying. (2) By using multi-versioning, each applied entry inserts a new revision into the MV storage.

In this section, we first provide background information on the MV storage in etcd, then present our pre-apply optimization. Lastly, we compare our method with log replay in DB replication.

Data model of etcd [18]. The database of etcd is a persistent MV key-value store. Unlike databases that perform in-place updates, the key-value pairs of etcd are immutable. A new update inserts a new **revision**, which grows monotonically over the cluster's lifetime. The key space maintains multiple revisions, starting from 1. A new revision is created when an atomic mutation is applied to the key space. For example, multiple keys updated by the same transaction share the same revision. Old data from previous revisions remains unchanged until garbage collection is triggered. Once GC is invoked, all revisions before a target revision are removed. Each key-value pair is also tagged with a **version**, but revisions increase cluster-wide, while versions increase key-wise.

RSM pre-apply in etcd. In the pre-apply optimization, when an entry is sent to the leader for ordering (L3 in Algo2), the following steps occur: (1) The leader optimistically applies the entry and generates revisions/versions for each key-value pair, although the revisions are not visible for reads. The leader can concurrently apply entries as long as each applying is atomic. The leader assigns the LSN based on the actual applied order. (2) As described in section 4, Fleet persists and replicates each entry, along with its LSN and revisions/versions. (3) Upon receiving the entries, each non-leader replica inserts the new revisions/versions into the MV storage, where these revisions are also not visible. (4) Each replica (leader included) maintains a visibility revision watermark (VRW), ensuring that read operations ignore all versions/revisions above VRW (i.e., search only begins from VRW). Thus, applying an entry (as per Rule 2 of algorithm 1) in this optimization barely updates the VRW, making it unlikely to become a performance bottleneck. Thus, the performance of the applying is significantly enhanced.

During leader election and recovery, the new leader identifies the latest applied LSN l and notifies the other RNodes. The etcd MVCC database on the other RNodes must remove revisions larger than l before updating the node's term. It is important to note that pre-apply is *not* a universal optimization for RSM, as it relies on storage that maintains immutable old versions. Thus, databases that perform in-place updates are not suitable for our optimizations. Additionally, consensus protocols that do not allow concurrent log entries (e.g., Raft) do not benefit from pre-apply.

Comparison with log replay on backup. The problem of applying the committed log resembles primary/backup replication with log shipping. Recent research [6, 22, 23] has proposed concurrent replay transactions with monotonic prefix consistency. However, pre-apply optimization differs in the following ways: (1) Works on parallel replay [6, 22, 23] rely on some form of dependency graph generated by the primary, such as dependencies among transactions or rows, and the backup must follow this dependency order during replay. Our technique does not require this dependency graph; a follower can apply an entry immediately upon receipt, as our consistency is provided by the visibility control of VRW. (2) The primary DB typically generates an ordered log, imposing order restrictions on transmission. In other words, the backup DB must receive log entries in the same order as the primary DB sends them. Our technique, using Fleet, does not have such restrictions and achieves high parallelism.

6 EVALUATION

We have integrated the proposed techniques into the code of etcd v3.5 and evaluated them on two types of distributed applications: an in-memory key-value store and the fully-featured etcd database (a persistent B-tree MV). Both applications ensure that all operations are executed in the same order across all replicas.

Given that this paper focuses on durable RSMs, our experiments do not include protocols based on the fail-stop model such as CMP [56] because they do not persist data to disk or lack clarity on recovery, making them unsuitable for our target scenarios.

6.1 Experimental Setup

6.1.1 Implementation. We used the code base of etcd [20] v3.5, which includes a well-established Raft [44, 45] implementation thoroughly studied by open-source communities [2] and widely used in real-world applications. To ensure an apples-to-apples comparison, our implementation reuses etcd's modules for network transport, disk I/O (e.g., WAL), and message processing. We have added Fleet related fields to the message structure and implemented our own message handling logic. Fleet asynchronously dumps ordered logs to local disks through (f+1) non-leader replicas. To ensure a fair comparison, Fleet follows the same batching behavior as the open-source etcd code, including aspects such as message processing, data persistence, and transaction group commitment. The pre-apply optimization has also been implemented in etcd's official database package (server/mvcc), which is a disk-backed MV B-tree.

We have also implemented Scalog within the same framework, using primary-backup replication and the "Mask" failure-handling strategy, which can automatically ignore failed nodes. This strategy is comparable to the quorum replication used by other competitors. Each node in this setup is equipped with *n log segments* (where n equals the replication factor) to increase parallelism. The correctness of Scalog requires that all backups receive the same order of operations within each log segment. This is achieved by using a single thread to copy the reference of operations into the queues (channels in Golang) for each of the backup nodes. Consequently, the actual transmission of operations can be handled by separate threads, enabling parallelism. This method exactly follows the open-source Scalog [50]. Additionally, a co-located aggregator summarizes a local cut every 0.1 milliseconds, as described in [15]. The ordering layer utilizes Raft, co-located within the same n nodes, to ensure data persistence.

Application. To evaluate Fleet, we use **MemKV** as the application layer (RSM state). The implementation of MemKV consists of multi-bucket in-memory hash tables. Each bucket is accessed using a hash function and protected by a read-write lock. A single *apply thread* sequentially executes write operations to ensure consistency, as multiple apply threads may violate this principle. To improve efficiency, we assign the apply thread to a dedicated OS thread and offload work such as serialization and deserialization to other threads whenever possible. To evaluate the efficiency of the pre-apply optimization, the consensus-committed log is applied to the **etcd**'s MV database (a disk-backed, B-tree-based structure).

6.1.2 Environment Settings. Our test environment consists of 15 virtual machines in the same availability zone of a public cloud. Each instance is equipped with 64 cores, 512 GB memory, 6 NVMe SSD drives with 1920GB capacity (Spec: write bandwidth 2 GB/s, 130k IOPS), and 20 Gbps network bandwidth. The OS is Linux kernel 5.10 LTS, and we used Golang v1.9 to generate executables. We evaluated the performance using two types of deployment layouts:

(1) For the scale-out and skewness experiments, we used separated deployment with 3 serving nodes and multiple storage groups, each consisting of 3 machines that can tolerate one failure.
(2) For all other experiments, we used a co-located deployment with 5 machines that can tolerate up to 2 failures, a typical setting that handles one unexpected failure along with one scheduled maintenance.

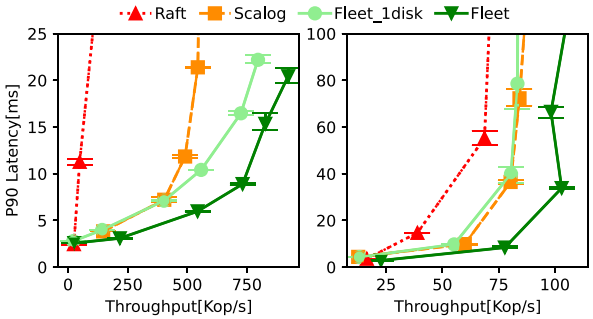


Figure 3: Throughput and Latency Comparison for 64B (Left) and 4KB (Right) Value Sizes

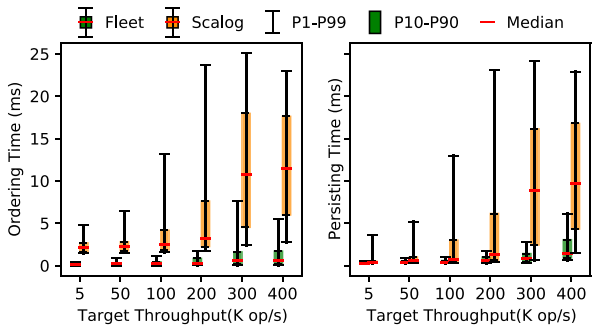


Figure 4: Performance Breakdown: Ordering Time (Left) and Persisting Time (Right) for Write Operations (Shared Y-Axis)

In co-located deployments, we optimize the setup by buffering entries to avoid repeatedly sending the same entry.

6.1.3 Workloads. To evaluate the performance of the MemKV, we employed workloads generated by YCSB [13]. The keyspace consists of 1 million keys, each with a size of 256 bytes, and values of either 64B or 4KB. We used a uniform key distribution unless otherwise stated. For etcd, we used the official etcd benchmark tools, with default key and value sizes of 8 bytes. All experiments focused primarily on write operations, running for 60 seconds, and we recorded results excluding the first and last 15 seconds. Each experiment was run 3 times, and we used vertical bars to indicate the maximum and minimum results.

6.2 Write Performance

Fleet and Scalog writes to all 6 local disks available on each machine. Raft server writes to a single local disk and cannot benefit from extra disks due to the ordering restriction. For completeness, we also include the performance of Fleet that writes to a single disk on each machine.

Figure 3 presents results obtained using both small and large values. Our findings indicate that Fleet achieves 1.5× higher throughput (113 Kops) for 4KB values and around 10× higher throughput (1008 Kops) for 64B values compared to Raft. Even when using a single disk, Fleet achieves 85 Kops for 4KB values and 800 Kops for 64B values, representing a 14% and 7× improvement respectively. Scalog also achieves 5.6× higher throughput than Raft for small values. The superior performance of Fleet is attributed to the increased parallelism enabled by its out-of-order and concurrent persistence layer. In particular, the left figure (small value cases) can be used to illustrate that using a "persistence-first and assign-LSN-later" approach (see §2.3) in Raft does not achieve the performance levels of Fleet. This is because the size of only the persisted entry reference and LSN mapping is comparable to the small value entries, leading us to predict that the performance will not exceed that of the small value Raft configuration. Furthermore, this scenario indicates that Raft's performance is restricted by the concurrency limitations inherent in the protocol, as evidenced by the fact that storage, network, and CPU utilization remain significantly below the thresholds of potential resource bottlenecks. Although both Scalog and Fleet benefit from increased parallelism, Scalog's performance is lower than Fleet across various settings due to limitations in its parallelism mechanisms. In the next subsection, we will analyze the reasons for these differences in detail.

Compared to Raft, Fleet also achieves *lower latency* in most cases. However, under extremely low client load (with 2 client threads), Fleet exhibits slightly higher p90 latency compared to Raft (0.97 ms vs 0.85 ms), which is negligible in most use cases.

6.3 Performance Profile

We further break down the performance of Fleet and Scalog, both of which support parallelism. We enhanced the code with profiling to record the time each operation (log entry) takes to (1) determine the LSN, referred to as **ordering time**, and (2) persist the log entry on replica nodes, referred to as **persisting time**. For Scalog, the ordering time is measured from when a log entry is written to disk until the LSN is received from the ordering layer. For Fleet, this refers to the round-trip message time of querying the leader for the LSN. Persisting time for both systems is the time spent on persisting the log entry on the replicas. The key difference between the two is that Fleet does not impose order constraints between log entries, whereas Scalog requires that each primary sends log entries to backups in a consistent order.

Figure 4 presents the results, illustrating the ordering time (left figure) and persisting time (right figure) under varying throughput levels. We provide a range of percentiles to offer a comprehensive depiction. It is important to note that the profiling introduces a slight overhead, resulting in absolute latency values that are marginally higher than those shown in Figure 3. Nevertheless, this overhead is consistent across both systems, rendering the relative values comparable. Under low system load (low throughput), Scalog's ordering time is noticeably higher than that of Fleet. This discrepancy arises because Scalog's ordering layer necessitates a complete Paxos/Raft consensus to determine the LSN, which is inherently more time-consuming than a simple leader query in Fleet. Although Scalog's ordering layer resolves the LSNs for all shards in a single consensus round, we observe that as system load increases, the ordering time correspondingly escalates. Under high load conditions, the P90 and P99 ordering times for Scalog are 9 times and 3.3 times higher than those for Fleet, respectively. Furthermore, the variability in Scalog's ordering time is substantially greater, as evidenced by the spread between the P1 and the P99. This increased variability is attributable to Scalog's requirement for coordination across all replicas of all shards, rendering it more vulnerable to tail latency and straggler effects.

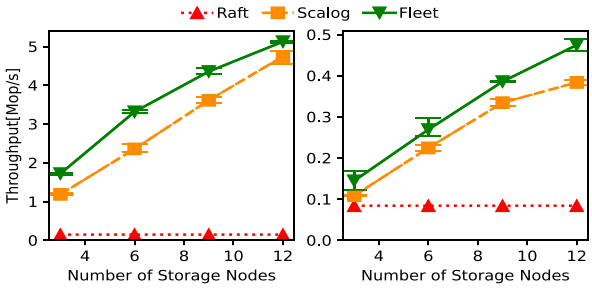
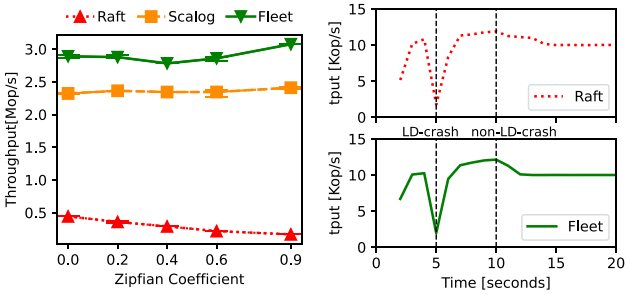


Figure 5: Scale-out performance for 64B (left) and 4KB (right) value sizes. The red dashed line shows the 3-node Raft result.



(a) Various Skewness(64B value) **(b) Recovery Experiment**

Figure 6: Various Skewness and Recovery Experiments.

More interestingly, the graph of persisting time contrasts the cost of enforcing ordering on the processes of replication and persisting. Under low system load, Scalog and Fleet exhibit similar median and P90 persisting times, but Scalog's P99 persisting time is noticeably higher. At higher throughput levels, Scalog's median and P90 persisting times also surpass those of Fleet. This observation indicates that enforcing strict ordering, as opposed to out-of-order execution, results in lower concurrency. Moreover, it is more susceptible to head-of-line blocking and tail latency effects, leading to greater latency variance under high system loads.

6.4 Scale Out

The purpose of this experiment is to examine the scalability of Fleet by adding more machines to the cluster. Clients are spawned only on machines that host storage nodes.

Figure 5 presents the results obtained using 64B values and 4KB values, respectively. Both figures illustrate how optimal throughput changes with the increasing number of storage nodes. For reference, each figure includes a horizontal line representing the optimal throughput of Raft using 3 replicas.

For 64B values, the performance of Fleet scales until it reaches the maximum speed at which the apply thread can execute put operations on the in-memory map, approximately 5 Mops. This is 35× Raft's 147 Kops. For 4KB values, Fleet achieves a throughput of 456 Kops, which is almost 5.4× Raft's 84 Kops. The bottleneck for Fleet lies in the network, as its performance scales linearly until the network bandwidth is saturated. When using 12 storage nodes, each replica consumes 2.4GBps of its 2.5GBps bandwidth, which is roughly the maximum throughput achievable by a no-replication state machine using this type of machine.

The results for Scalog confirm that it can scale out nearly linearly but achieves slightly lower throughput than Fleet. We attribute the performance difference to the additional work Scalog performs in generating and transmitting local cuts and global decisions periodically (every 0.1 ms). Besides, as previously discussed, the scale-out results of the Fleet have reached the maximum replication capacity. Consequently, any additional scaling of Scalog will only approach this limit, as Fleet merely converges on the threshold.

6.5 Comparison with Partitioned Clusters

Classical consensus protocols cannot scale effectively by adding more machines. However, partitioning states into shards allows scaling by assigning each an independent RSM. This section compares the performance of this approach to the performance of Fleet in a realistic scenario: a spiky workload originating from a small portion of applications. The goal is to determine if the peak throughput remains within acceptable limits when the workloads concentrate on a small range of keyspace.

In this experiment, we use a total of 9 machines for each setup. For Raft, we evenly divide the keyspace into three shards, each managed by a Raft group with 3 replicas. For Fleet and Scalog, we use the separated deployment type with 3 replicas and 6 storage nodes. It is important to note that we do not partition the keyspace for Fleet and Scalog, as they apply all commands in a total order. In contrast, Raft only provides weaker consistency since there is no ordering constraint across shards.

The key access distribution follows a Zipfian distribution with ζ ranging from 0 to 0.9. When $\zeta = 0$, the distribution is equivalent to the uniform distribution. As ζ increases, the distribution becomes more skewed, such that requests disproportionately target one of the shards. For each value of ζ, we gradually increase the number of spawned client threads until the throughput converges.

Figure 6a presents the results obtained using 64B values. Even when the key access distribution is uniform ($\zeta = 0$), Fleet achieves more than 6.5× the throughput using the same number of machines despite the weaker consistency of multiple Raft groups. This is because each Raft group still faces limited parallelism due to the aforementioned ordering restriction. At the highest skew level ($\zeta = 0.9$), Fleet outperforms Raft by an even larger margin.

Figure 6a also demonstrates that the performance of multiple Raft groups declines as ζ increases, because an increasing number of requests are sent to the same Raft group. In contrast, Fleet maintains stable throughput because the workload can be distributed across the entire cluster. Additionally, Fleet maintains 25% higher throughput than Scalog, as observed in previous experiments.

6.6 Failure Recovery

This experiment aims to evaluate the performance of Fleet and Raft under failure conditions. Since Scalog also uses Raft as its fault-tolerance layer, its behavior is similar to that of Raft (e.g., Figure 4 in Scalog [15]). Therefore, we excluded Scalog from this subsection.

The YCSB target throughput is set at 10,000 ops, a rate at which neither system is saturated and both maintain steady throughput. We intentionally kill the leader and a non-leader replica at times (t = 5) seconds and (t = 10) seconds, respectively. The heartbeat interval plays a crucial role in failure detection, as false suspicions of failure can negatively impact performance. For our evaluation, we use the heartbeat interval value suggested by etcd: 100 ms,

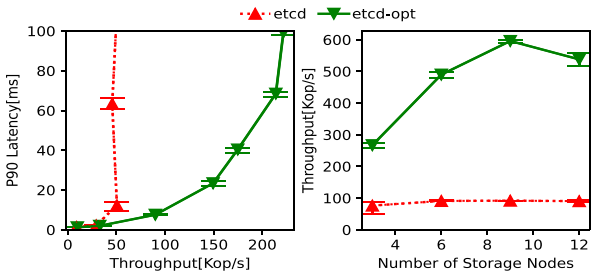


Figure 7: Left: Throughput vs. Latency of etcd Benchmark. Right: etcd Scale-Out Experiments. etcd-opt uses pre-apply.

with an election triggered after 10 missed heartbeats. YCSB reports the throughput at its minimum interval of 1 second. In addition to throughput, we measure the time taken for the new leader to become fully functional.

Based on 100 runs, Raft takes 1126 ms for recovery after a leader crash, while Fleet takes 1175 ms. The additional 50 ms overhead (4.5% increase) is acceptable given the infrequency of recovery operations. This overhead has minimal impact on throughput, as demonstrated by Figure 6b. Both Raft's and Fleet's throughput drop at similar rates after the leader crashes and resume to the target throughput around the same time after the new leader is elected and becomes functional. Furthermore, we observe that non-leader crashes have no impact on throughput for either system.

6.7 Experiment of etcd

To evaluate the improvement in parallelism achieved by pre-applying compared to the original sequential apply, we use Fleet as the underlying log replication technique and the etcd MV database as the application layer.

In Figure 7, the left figure illustrates the results of throughput and latency using 5 nodes. Pre-apply outperforms sequential apply in both metrics. Even under extremely low client load (with 2 client threads), sequential apply exhibits 14% higher 90th percentile latency (1.46 ms vs. 1.27 ms). Meanwhile, pre-applying achieves a peak throughput that is 4.4× higher than sequential apply.

The right figure shows the results of scale-out experiments, with settings the same as in §6.4. We observe that sequential apply has stable throughput as it already reaches its peak performance. On the other hand, pre-applying enables throughput to grow nearly linearly until 9 storage nodes (a total 12-node cluster), achieving a peak throughput of around 600K ops—more than 10× higher than sequential apply. However, when 12 storage nodes are used, the leader becomes overloaded in this implementation, resulting in a slight performance decline.

7 RELATED WORK

Coordination services can be implemented using the abstraction of RSMs [33], which rely on consensus protocols to coordinate replicas to agree on the same sequence of deterministic commands. According to the FLP impossibility theorem [19], consensus is not solvable in an asynchronous setting even with a single failure.

Paxos [34, 35, 38, 55] is one of the most influential consensus protocols, inspiring a series of works [32, 36, 37, 42, 43]. Despite their significant influence in academia, the Paxos/Multi-Paxos protocols are challenging to implement in practice due to their lack of clarity in technical details. Raft [44, 45], a more recent protocol, addresses this gap by introducing a more comprehensible approach, which has gained substantial traction in industry [20, 25, 45, 52]. Both Raft and Paxos are leader-based protocols susceptible to the single-leader bottleneck, and several works have attempted to mitigate this issue [12, 37]. Various systems have been proposed to enhance the performance. Some improve read throughput by allowing reads from followers [11, 54], while others enhance write throughput by distributing writes to shards in a round-robin manner [4, 42]. Additional approaches include persisting entries before ordering them [8, 15, 43], using weaker consistency or persistence guarantees [30, 32], or offloading consensus functions to new network hardware [28, 39]. Nonetheless, none of these solutions satisfy all five requirements for coordination services as specified in § 2.1.

Whittaker et al. [56] analyze the performance bottlenecks of consensus protocols and apply their compartmentalization technique to MultiPaxos. The resulting Compartmentailized MultiPaxos (CMP) outperforms MultiPaxos at the expense of more machines. CMP uses grid quorum [24, 46], a type of flexible quorum that organizes nodes into a grid layout. Furthermore, CMP is completely in-memory while Fleet persists all data to disk to further reduce the chance of data loss.

Bessani et al. [7] analyzed the performance problems of durable RSMs and proposed parallel logging, which aims to batch loggings from different consensus groups to improve efficiency when writing to disk. This approach is orthogonal to Fleet's enhancements in the concurrency within a single consensus group. Eventually Durable (ED) RSM [31] allows applications to make latency/durability tradeoffs but entails the risk of data loss, making it unsuitable for our target scenarios.

8 CONCLUSION

This paper identifies several bottlenecks in existing coordination services and proposes Fleet, a high-performance RSM protocol that leverages a hybrid log approach to enhance performance. Fleet decouples the fast persistence of individual log entries from the requirement for totally ordered persistence. By relaxing ordering and placement restrictions, Fleet achieves higher parallelism without compromising consistency. It achieves fault tolerance through a hybrid storage layer that integrates scattered log entries for synchronous persistence and centralized ordered logs for asynchronous persistence, facilitating rapid recovery. Additionally, Fleet incorporates a parallel applying optimization for etcd.

Our evaluation demonstrates that Fleet significantly enhances throughput without compromising latency, achieving a 7× increase in throughput compared to Raft on identical hardware. With additional resources, Fleet can scale linearly until constrained by network bandwidth or single-thread processing limitations. Enabled with the pre-apply optimization in etcd, it achieves 600k operations per second using a 12-node cluster, which is 10× faster than the baseline. Although Fleet incurs a 50ms delay in recovery time following leader failures, the impact is minimal due to the infrequency of such events.

REFERENCES

[1] Ailidani Ailijiang, Aleksey Charapko, and Murat Demirbas. 2016. Consensus in the Cloud: Paxos Systems Demystified. In *2016 25th International Conference on Computer Communication and Networks (ICCCN)*. 1–10. https://doi.org/10.1109/ICCCN.2016.7568499

[2] Raft authors. 2024. Raft Consensus Algorithm. https://raft.github.io/. Accessed: 2024-11-01.

[3] The Kubernetes authors. 2024. Kubernetes: Production-Grade Container Orchestration. https://kubernetes.io. Accessed: 2024-11-01.

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

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

[6] MariaDB Knowledge Base. 2024. Parallel Replication. https://mariadb.com/kb/en/parallel-replication. Accessed: 2024-11-01.

[7] Alysson Bessani, Marcel Santos, João Felix, Nuno Neves, and Miguel Correia. 2013. On the Efficiency of Durable State Machine Replication. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX Association, San Jose, CA, 169–180. https://www.usenix.org/conference/atc13/technical-sessions/presentation/bessani

[8] Martin Biely, Zarko Milosevic, Nuno Santos, and André Schiper. 2012. S-Paxos: Offloading the Leader for High Throughput State Machine Replication. In *2012 IEEE 31st Symposium on Reliable Distributed Systems*. 111–120. https://doi.org/10.1109/SRDS.2012.66

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

[10] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 275–290. https://doi.org/10.1145/3183713.3196898

[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 19)*. USENIX Association, Renton, WA. https://www.usenix.org/conference/hotstorage19/presentation/charapko

[12] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. 2021. *PigPaxos: Devouring the Communication Bottlenecks in Distributed Consensus*. Association for Computing Machinery, New York, NY, USA, 235–247. https://doi.org/10.1145/3448016.3452834

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

[14] Medhavi Dhawan, Gurprit Johal, Jim Stabile, Vjekoslav Brajkovic, James Chang, Kapil Goyal, Kevin James, Zeeshan Lokhandwala, Anny Martinez Manzanilla, Roger Michoud, Maithem Munshed, Srinivas Neginhal, Konstantin Spirov, Michael Wei, Scott Fritchie, Chris Rossbach, Ittai Abraham, and Dahlia Malkhi. 2017. Consistent Clustered Applications with Corfu. *SIGOPS Oper. Syst. Rev.* 51, 1 (sep 2017), 78–82. https://doi.org/10.1145/3139645.3139658

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

[16] etcd authors. 2024. etcd mvcc package. https://github.com/etcd-io/etcd/tree/v3.5.1/server/mvcc. Accessed: 2024-11-01.

[17] etcd authors. 2024. etcd mvcc server. https://github.com/etcd-io/etcd/tree/v3.5.1/server. Accessed: 2024-11-01.

[18] etcd docs. 2024. Data model. https://etcd.io/docs/v3.5/learning/data_model/. Accessed: 2024-11-01.

[19] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (April 1985), 374–382.

[20] Cloud Native Computing Foundation. 2024. A distributed, reliable key-value store for the most critical data of a distributed system. https://etcd.io/. Accessed: 2024-11-01.

[21] C. Gray and D. Cheriton. 1989. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles (SOSP '89)*. Association for Computing Machinery, New York, NY, USA, 202–210. https://doi.org/10.1145/74850.74870

[22] Jeffrey Helt, Abhinav Sharma, Daniel J. Abadi, Wyatt Lloyd, and Jose M. Faleiro. 2022. C5: Cloned Concurrency Control That Always Keeps Up. *Proc. VLDB Endow.* 16, 1 (sep 2022), 1–14. https://doi.org/10.14778/3561261.3561262

[23] Chuntao Hong, Dong Zhou, Mao Yang, Carbo Kuo, Lintao Zhang, and Lidong Zhou. 2013. KuaFu: Closing the parallelism gap in database replication. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 1186–1195. https://doi.org/10.1109/ICDE.2013.6544908

[24] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. 2017. Flexible Paxos: Quorum Intersection Revisited. In *20th International Conference on Principles of Distributed Systems (OPODIS 2016) (Leibniz International Proceedings in Informatics (LIPIcs))*, Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone (Eds.), Vol. 70. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 25:1–25:14. https://doi.org/10.4230/LIPIcs.OPODIS.2016.25

[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 (aug 2020), 3072–3084. https://doi.org/10.14778/3415478.3415535

[26] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Boston, MA) *(USENIXATC'10)*. USENIX Association, USA, 11.

[27] Michael Isard. 2007. Autopilot: Automatic Data Center Management. *SIGOPS Oper. Syst. Rev.* 41, 2 (apr 2007), 60–67. https://doi.org/10.1145/1243418.1243426

[28] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. 2016. Consensus in a Box: Inexpensive Coordination in Hardware. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 425–438. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/istvan

[29] Andrew Jeffery, Heidi Howard, and Richard Mortier. 2021. Rearchitecting Kubernetes for the Edge. In *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking* (Online, United Kingdom) *(EdgeSys '21)*. Association for Computing Machinery, New York, NY, USA, 7–12. https://doi.org/10.1145/3434770.3459730

[30] Tian Jiang, Xiangdong Huang, Shaoxu Song, Chen Wang, Jianmin Wang, Ruibo Li, and Jincheng Sun. 2023. Non-Blocking Raft for High Throughput IoT Data. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 1140–1152. https://doi.org/10.1109/ICDE55515.2023.00092

[31] Kriti Kathuria. 2024. *Eventually Durable State Machines*. Master's thesis. University of Waterloo, Waterloo, Canada.

[32] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. 2013. MDCC: Multi-Data Center Consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic) *(EuroSys '13)*. Association for Computing Machinery, New York, NY, USA, 113–126. https://doi.org/10.1145/2465351.2465363

[33] Leslie Lamport. 1984. Using Time Instead of Timeout for Fault-Tolerant Distributed Systems. *ACM Trans. Program. Lang. Syst.* 6, 2 (April 1984), 254–280.

[34] Leslie Lamport. 1998. The Part-time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133–169. https://doi.org/10.1145/279227.279229

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

[36] Leslie Lamport. 2005. *Generalized Consensus and Paxos*. Technical Report MSR-TR-2005-33. 60 pages. https://www.microsoft.com/en-us/research/publication/generalized-consensus-and-paxos/

[37] Leslie Lamport. 2006. Fast Paxos. *Distributed Computing* 19 (October 2006), 79–103. https://www.microsoft.com/en-us/research/publication/fast-paxos/

[38] Butler Lampson. 1996. How to Build a Highly Available System Using Consensus. In *10th International Workshop on Distributed Algorithms (WDAG 1996)*, Ozalp Babaoglu and Keith Marzullo (Eds.). Springer, 1–17. https://www.microsoft.com/en-us/research/publication/how-to-build-a-highly-available-system-using-consensus/ The proceedings are: Distributed Algorithms, Lecture Notes in Computer Science 1151, Springer, 1996.

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

[40] Tianyu Li, Badrish Chandramouli, Jose M. Faleiro, Samuel Madden, and Donald Kossmann. 2021. Asynchronous Prefix Recoverability for Fast Distributed Stores. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1090–1102. https://doi.org/10.1145/3448016.3458454

[41] Barbara Liskov and James Cowling. 2012. *Viewstamped Replication Revisited*. Technical Report MIT-CSAIL-TR-2012-021. MIT.

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

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

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

[45] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 305–319. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[46] Marta Patiño-Martinez. 2009. *Quorum Systems*. Springer US, Boston, MA, 2317–2317. https://doi.org/10.1007/978-0-387-39940-9_299

[47] Marius Poke and Torsten Hoefler. 2015. DARE: High-Performance State Machine Replication on RDMA Networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing* (Portland, Oregon, USA) *(HPDC '15)*. Association for Computing Machinery, New York, NY, USA, 107–118. https://doi.org/10.1145/2749246.2749267

[48] Stefanos Sagkriotis and Dimitrios Pezaros. 2022. Accelerating Kubernetes with In-Network Caching. In *Proceedings of the SIGCOMM '22 Poster and Demo Sessions* (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 40–42. https://doi.org/10.1145/3546037.3546058

[49] Stefanos Sagkriotis and Dimitrios Pezaros. 2023. Scalable Data Plane Caching for Kubernetes. In *Proceedings of the 18th International Conference on Network and Service Management* (Thessaloniki, Greece) *(CNSM '22)*. International Federation for Information Processing, Laxenburg, AUT, Article 66, 7 pages.

[50] scalog authors. 2024. scalog. https://github.com/scalog/scalog. Accessed: 2024-11-01.

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

[52] 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 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1493–1509. https://doi.org/10.1145/3318464.3386134

[53] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutornenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. 2020. Twine: A Unified Cluster Management System for Shared Infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 787–803. https://www.usenix.org/conference/osdi20/presentation/tang

[54] Jeff Terrace and Michael J. Freedman. 2009. Object Storage on CRAQ: High-Throughput Chain Replication for Read-Mostly Workloads. In *2009 USENIX Annual Technical Conference (USENIX ATC 09)*. USENIX Association, San Diego, CA. https://www.usenix.org/conference/usenix-09/object-storage-craq-high-throughput-chain-replication-read-mostly-workloads

[55] Robbert Van Renesse and Deniz Altinbuken. 2015. Paxos Made Moderately Complex. *ACM Comput. Surv.* 47, 3, Article 42 (feb 2015), 36 pages. https://doi.org/10.1145/2673577

[56] Michael Whittaker, Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, Neil Giridharan, Joseph M. Hellerstein, Heidi Howard, Ion Stoica, and Adriana Szekeres. 2021. Scaling Replicated State Machines with Compartmentalization. *Proc. VLDB Endow.* 14, 11 (jul 2021), 2203–2215. https://doi.org/10.14778/3476249.3476273

[57] Jie Zhang, Chen Jin, YuQi Huang, Li Yi, Yu Ding, and Fei Guo. [n.d.]. KOLE: breaking the scalability barrier for managing far edge nodes in cloud. In *Proceedings of the 13th Symposium on Cloud Computing* (San Francisco California, 2022-11-07). ACM, 196–209. https://doi.org/10.1145/3542929.3563462