Check for updates

Exploiting Nil-Externality for Fast Replicated Storage

Aishwarya Ganesan
VMware Research

Ramnatthan Alagappan
VMware Research

Andrea C. Arpaci-Dusseau
University of Wisconsin – Madison

Remzi H. Arpaci-Dusseau
University of Wisconsin – Madison

Abstract

Do some storage *interfaces* enable higher performance than others? Can one identify and exploit such interfaces to realize high performance in storage systems? This paper answers these questions in the affirmative by identifying *nil-externality*, a property of storage interfaces. A nil-externalizing (nilext) interface may modify state within a storage system but does not externalize its effects or system state immediately to the outside world. As a result, a storage system can apply nilext operations lazily, improving performance.

In this paper, we take advantage of nilext interfaces to build high-performance replicated storage. We implement Skyros, a nilext-aware replication protocol that offers high performance by deferring ordering and executing operations until their effects are externalized. We show that exploiting nil-externality offers significant benefit: for many workloads, Skyros provides higher performance than standard consensus-based replication. For example, Skyros offers 3× lower latency while providing the same high throughput offered by throughput-optimized Paxos.

CCS Concepts: • **Information systems** → **Distributed storage**; **Storage replication**.

Keywords: Fault-tolerance, Replication, Storage

1 Introduction

Defining the right interfaces is perhaps the most important aspect of system design [46], as well-designed interfaces often lead to desirable properties. For example, idempotent interfaces make failure recovery simpler [13, 70]; commutative interfaces enable scalable software implementations [14].

In a similar spirit, this paper asks: Do some types of interfaces enable higher performance than others in storage systems? Our exercise in answering this question has led us to identify an important storage-interface property which we call *nil-externality*. A nil-externalizing (nilext) interface may modify state within a storage system but does not externalize its effects or system state immediately to the outside world (apart from the acknowledgment itself). As a result, a storage system can apply a nilext operation in a deferred manner after acknowledgment, improving performance.

In this paper, we exploit nil-externality to design high-performance replicated storage that offers strong consistency (i.e., linearizability [36]). A standard approach today to building such a system is to use a consensus protocol like Paxos [44], Viewstamped Replication (VR) [52], or Raft [62]. For example, Facebook's ZippyDB uses Paxos to replicate RocksDB [73]; Harp builds a replicated file system using VR [53]; other examples exist as well [7, 17, 18, 22].

A storage system built using this standard approach performs several actions before it returns a response to a request. Roughly, the system makes the request durable (if it is an update), orders the request with respect to other requests, and finally executes the request. Usually, a leader replica orchestrates these actions [52, 62]. Upon receiving requests, the leader decides the order and then replicates the requests (in order) to a set of followers; once enough followers respond, the leader applies the requests to the system state and returns responses. Unfortunately, this process is expensive: updates incur two round trips (RTTs) to complete.

The system can defer some or all of these actions to improve performance. Deferring durability, however, is unsafe: if an acknowledged write is lost, the system would violate linearizability [31, 48]. Fortunately, durability can be ensured without coordination: clients can directly store updates in a single RTT on the replicas [64, 80]. However, ordering (and subsequent execution) requires coordination among the replicas and thus is expensive. Can a system hide this cost by deferring ordering and execution?

At first glance, it may seem like all operations must be synchronously ordered and executed before returning a response. However, we observe that if the operation is nilext, then it can be ordered and executed lazily because nilext operations do not externalize state or effects immediately.

Nilext interfaces have performance advantages, but are they practical? Perhaps surprisingly, we find that nilext interfaces are not just practical but prevalent in storage systems (§2). As a simple example, consider the put interface in the key-value API. Put is nilext because it does not externalize

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SOSP '21, October 26–29, 2021, Virtual Event, Germany
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8709-5/21/10…$15.00
https://doi.org/10.1145/3477132.3483543

the state of the key-value store: it does not return an execution result or an execution error (for instance, by checking if the key already exists). In fact, popular key-value stores such as RocksDB [29], LevelDB [33], and others built atop write-optimized structures (like LSMs [63] and B$^{\epsilon}$-trees [8]) transform *all* updates into nilext writes by design; querying a write-optimized structure before every update can be very expensive [6]. Thus, in these systems, even updates that read prior state and modify data are nilext (in addition to blind writes that simply overwrite data).

Nilext-aware replication is a new approach to replication that takes advantage of nil-externality of storage interfaces (§3). The key idea behind this approach is to defer ordering and executing operations until their effects are externalized. Because nilext updates do not externalize state, they are made durable immediately, but expensive ordering and execution are deferred, improving performance. The effects of nilext operations, however, can be externalized by *later* non-nilext operations (e.g., a read to a piece of state modified by a nilext update). Thus, nilext operations must still be applied in the same (real-time) order across replicas for consistency. This required ordering is established in the background and enforced before the modified state is externalized. While nilext interfaces lead to high performance, it is, of course, impractical to make all interfaces nilext: applications do need state-externalizing updates (e.g., increment and return the latest value, or return an error if key is not present). Such non-nilext updates are immediately ordered and executed for correctness.

Nilext-aware replication delivers high performance in practice. First, while applications do require non-nilext updates, such updates are less frequent than nilext updates. For instance, nilext set is the most popular kind of update in Memcached [1]. Similarly, put, delete, and merge (read-modify-writes that do not return results), which are all nilext, are the dominant type of updates in ZippyDB [11]. We find similar evidence in production traces from IBM [24] and Twitter [79]. Further, while reads do externalize state, not every read triggers synchronous ordering. In many workloads, updates to an object can be ordered and executed in the background before applications read the object. Our analyses of production traces from IBM COS [24] reveal that this is indeed the case (§3.3).

Nilext-aware replication draws inspiration from the general idea of deferring work until needed similar to lazy evaluation in functional languages [37], externally synchronous file I/O [60], and previous work in databases [30, 68]. Here, we apply this general idea to hide the cost of ordering and execution in replicated storage. Prior approaches like speculative execution [41, 42, 67] reduce ordering cost by eagerly executing and then verifying that the order matches before notifying end applications. Nilext-aware replication, in contrast, realizes that some operations can be lazily ordered and executed *after* notifying end applications of completion.

We build SKYROS, a new protocol that adapts state machine replication [71] to take advantage of nilext interfaces (§4). The main challenge in our design is to ensure linearizability (especially during view changes) while maintaining high performance. To this end, SKYROS applies many techniques. SKYROS first uses supermajority quorums and a new durability-log design to complete nilext writes in one RTT. Second, SKYROS implements an ordering-and-execution check to serve reads in one RTT. Finally, SKYROS employs a DAG-based order-resolution technique to reconstruct the linearizable order during view changes.

While SKYROS defers ordering, Generalized Paxos [45], Curp [64], and other protocols [58, 65] realize that ordering is in fact not needed when operations commute. However, these protocols incur overhead when writes conflict and when interface operations do not commute. For instance, when multiple writers append records to a file (a popular workload in GFS [32]), these protocols incur high overhead (2 or 3 RTTs in Curp). In contrast, SKYROS can defer ordering such operations because they are nilext. More importantly, nil-externality is compatible with commutativity: a nilext-aware protocol can also exploit commutativity to quickly commit non-nilext updates. We build SKYROS-COMM, a variant of SKYROS to demonstrate this compatibility.

Our experiments (§5) show that SKYROS offers 3× higher throughput than Paxos (without batching) for a nilext-only workload. While batching improves Paxos' throughput, at peak throughput, SKYROS offers 3.1× lower latency. We run extensive microbenchmarks, varying request ratios, distributions, and read-latest fractions. SKYROS outperforms Paxos (with batching) in most cases; even when pushed to extremes (e.g., all non-nilext writes), SKYROS performs as well as Paxos. Under write-heavy YCSB workloads, SKYROS is 1.4× to 2.3× faster. For read-heavy workloads, while throughput gains are marginal, SKYROS reduces p99 latency by 70%. We also use SKYROS to replicate RocksDB with high performance. Finally, we compare SKYROS to Curp [64], a recent commutative protocol. Curp performs well (like SKYROS) when operations commute. However, when operations do not commute but are nilext, SKYROS offers advantages: SKYROS provides 2× better throughput for file record appends and 2.7× lower p99 latency in a key-value store. SKYROS-COMM combines the best of both worlds: it quickly completes nilext operations and exploits commutativity to speedup non-nilext operations.

This paper makes four contributions.

- We first identify nil-externality, a property of storage interfaces, and show its prevalence.
- We show how one can exploit this property to improve the performance of strongly consistent storage systems.
- Third, we present the design and implementation of SKYROS, a nilext-aware replication protocol.
- Finally, we demonstrate the performance benefits of SKYROS through rigorous experiments.

2 Nil-Externalizing Interfaces

We first define nil-externality and describe its attributes. We next analyze which interfaces are nilext in three example storage systems; then, we discuss opportunities to improve performance by exploiting nilext interfaces in general.

2.1 Nil-externality

We define an interface to be nil-externalizing if it does not externalize storage-system state: it does not return an execution result or an execution error, although it might return an acknowledgment. A nilext interface can modify state in any way (blindly set, or read and modify). The state modified by a nilext operation can be externalized at a later point by another non-nilext operation (e.g., a read). Note that although nilext operations do not return an *execution* error, they may return a *validation* error. Validation errors (e.g., a malformed request) do not externalize state and can be detected without executing the operation. Thus, an operation that returns only validation errors (but not execution errors) is nilext.

Determining whether or not an operation is nilext is simple in most cases. Nil-externality is an interface-level property: it suffices to look at the interface (specifically, the return value and the possible execution errors) to say if an operation is nilext. Nil-externality is a static property: it is independent of the system state or the arguments of an operation; one can therefore determine if an operation is nilext without having to reason about all possible system states and arguments.

2.2 Nil-externality in Storage Systems

We now analyze which interfaces are nilext in three storage systems that expose a key-value API (see Table 1). We pick these systems as candidates given their widespread use [11, 27, 55, 61]; exploiting nilext interfaces in these systems to improve performance can benefit many deployments.

RocksDB and LevelDB are LSM-based [63] key-value stores. Put in these systems is a nilext interface: it does not return an execution result or an error by checking record-existence. Similarly, write (multi-put) is also nilext. Delete is nilext because it does not return an error if the key is not present; it simply inserts a tombstone for the key. Surprisingly, even read-modify-writes (RMW) are nilext. RocksDB supports RMW via the merge operator [28], which is implemented as an *upsert* [6]. An upsert encodes a modification by specifying a key k and a function F that transforms the value of k. In RocksDB and other stores [15, 33] built upon write-optimized structures (LSMs and B$^{\epsilon}$-trees), reading the value of a key before updating it is expensive [6, 11, 28]. Thus, an upsert is not immediately applied, but the function and the key are simply recorded. Since an upsert is not applied immediately, it does not return an execution result or an execution error and thus merge is nilext. In fact, all modifications in write-optimized stores are a form of upserts that avoid querying before updates [6], and thus are all nilext; for instance, the tombstone inserted upon a delete is an upsert. Finally, get externalizes system state and so is not nilext.

System	Update		Read
	Nilext	Non-nilext	
RocksDB	put,write, delete,merge		get,multiget
LevelDB	put,write, delete		get,multiget
Memcached	set	add^e,delete^e,cas^r,replace^e, append^e,decr^r,incr^r,prepend^e	get,gets

Table 1. Nil-externality in Storage Systems. *The table shows which operations are nilext in popular key-value systems.* I^e *denotes that update interface I is non-nilext because it returns an execution error (e.g., key not found);* I^r *denotes a non-nilext update that returns an execution result.*

In Memcached, set is nilext because it does not return an execution result or an error; all other update interfaces are non-nilext. However, as we soon show (§3.3), these non-nilext updates are used only rarely compared to nilext set.

Nilext updates can be completed faster than non-nilext ones because their ordering and execution can be deferred. Thus, operations such as put and set in the above systems can be completed quickly, improving performance. What such opportunities exist across storage systems in general? A typical storage system supports three kinds of operations: reads, writes, and RMWs [10, 76]. While reads are non-nilext, writes and RMWs can be further classified based on whether or not they externalize state. Thus, some writes are nilext (e.g., RocksDB put), while others are not (e.g., Memcached add); similarly, some RMWs are nilext (e.g., RocksDB merge), while some are not (e.g., Memcached incr). A system can lazily apply all such nilext updates to improve performance.

Note that while nilext operations do not return errors as part of their contract, a system that lazily applies nilext writes may encounter errors (e.g., due to insufficient disk space or a bad block) at a later point. A storage system that eagerly applies updates can detect such errors early on. Fortunately, this difference is not an obstacle to realizing the benefits of nilext interfaces in practice as we discuss later (§4.8).

Given the benefits of nilext interfaces, it is worthwhile to make small changes to a non-nilext interface's semantics to make it nilext when possible. For instance, a Btree-based store may return an error upon an update to a nonexistent key; changing the semantics to not return such an error can enable a system to replicate updates quickly. Such semantic changes have been practical and useful in the past: MySQL-TokuDB supports SQL updates that do not return the number of records affected to exploit TokuDB's fast upserts [66].

3 Nilext-aware Replication

We now describe how a replicated storage system can exploit nil-externality to improve performance. To do so, we first give background on consensus, a standard substrate upon which strongly consistent storage is built. We then describe the nilext-aware replication approach and show that its high-performance cases are common in practice. We finally discuss how this new approach compares to existing approaches.

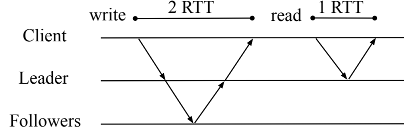


Figure 1. Request Processing in Consensus. *The figure shows how writes and reads are processed in systems built atop consensus protocols.*

3.1 Consensus-based Replication Background

Consensus protocols (e.g., Paxos, VR) ensure that replicas execute operations in the same order. Clients submit operations to the leader which then ensures that replicas agree on a consistent ordering of operations before executing them.

Figure 1 shows how requests are processed in the failure-free case. Upon an update, the leader assigns an index, adds the request to its log, and sends a *prepare* to the followers. The followers add the request to their logs and respond with a *prepare-ok*. Once the leader receives *prepare-ok* from enough followers, it applies the update and returns the result to the client. Reads are usually served by the leader locally; the leader is guaranteed to have seen all updates and so can serve the latest data, preserving linearizability. Stale reads on a deposed leader can be prevented using leases [52].

Latency is determined by the message delays in the protocol: updates take two RTTs and reads one RTT. Throughput is determined by the number of messages processed by the leader [21]. Practical systems [3] batch requests to reduce the load on the leader. While batching improves throughput, it increases latency, a critical concern for applications [67, 69].

3.2 Exploiting Nil-externality for Fast Replication

Using an off-the-shelf consensus protocol to build replicated storage leads to inefficiencies because this approach is oblivious to the properties of the storage interface. In particular, it is oblivious to nil-externality: *all* updates are immediately ordered and executed. Our hypothesis is that a replication protocol can deliver higher performance if it is cognizant of the underlying storage interface. Specifically, if a protocol is aware of nil-externality, it can delay ordering and execution, improving performance. We now provide an overview of such a protocol. We describe the detailed design soon (§4).

A nilext-aware protocol defers ordering and execution of operations until their effects are externalized. Figure 2 shows how such a protocol handles different operations. First, nilext writes are made durable immediately, but their ordering and execution are deferred. Clients send nilext writes to all replicas. Clients wait for enough replies including one from the leader before they consider the request to be completed. Nilext writes thus complete in one RTT. At this point, the operation is durable and considered *complete*; clients can make progress without waiting for the operation to be ordered and executed. We say that an operation is *finalized* when it is assigned an index and applied to the storage system.

State modified by nilext updates can be externalized later by other non-nilext operations (e.g., reads). Therefore, the

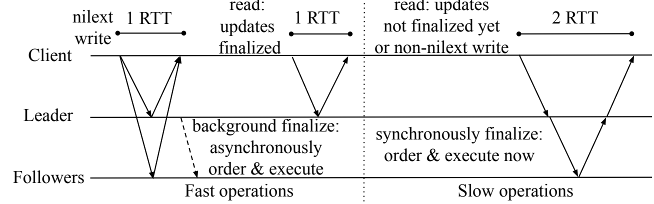


Figure 2. Nilext-aware Replication. *The figure shows how a nilext-aware replication protocol handles different operations.*

protocol must ensure that replicas apply the updates in the same order and it has to do so before the modifications are externalized. Thus, upon receiving a read, the leader checks if there are any unfinalized updates that this read depends upon. If no, it quickly serves the read. Conversely, if there are unfinalized updates, the leader synchronously establishes the order and waits for enough followers to accept the order; the leader then applies the pending updates and serves the read. In practice, most reads can be served without triggering synchronous ordering and execution because the leader keeps finalizing updates in the background; thus, in most cases, updates are finalized already by the time a read arrives.

Finally, the protocol does not defer ordering and executing non-nilext updates. Clients submit non-nilext requests to the leader which finalizes the request by synchronously ordering and executing it (and the previously completed requests).

A nilext-aware protocol can complete nilext updates in one RTT; non-nilext updates take two RTTs. A read can be served in one RTT if prior nilext updates that the read depends upon are applied before the read arrives. Thus, exploiting nil-externality offers benefit if a significant fraction of updates is nilext and reads do not immediately follow them. We next show that these conditions are prevalent in practice.

3.3 Fast Case is the Common Case

We first analyze the prevalence of nilext updates. First, we note that in some systems, almost all updates are nilext (e.g., write-optimized key-value stores as shown in Table 1). Some systems like Memcached have many non-nilext interfaces. However, how frequently do applications use them? To answer this question, we examine production traces [75, 79] from Twemcache, a Memcached clone at Twitter [74]. The traces contain ~200 billion requests across 54 clusters. Twemcache supports 9 types of updates (similar to Memcached as shown in Table 1). Except for set, others are non-nilext.

We consider 29 clusters that have at least 10% updates. Figure 3(a) shows the distribution of nilext percentages. In Twemcache, in 80% of the clusters, more than 90% of updates are nilext (set). This aligns with Memcached's expected usage [1] that most updates are sets and others are only sparingly used. Also, among the eight non-nilext updates, applications used only five: add, cas, delete, incr, and prepend. Among these, only incr and cas return an execution result, while others return execution errors; perhaps changing the

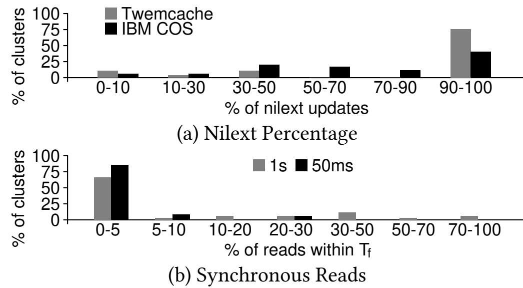


Figure 3. Fast Case is Common. *(a) shows the distribution of nilext percentages; a bar for a range x%-y% shows the percentage of clusters where x%-y% of updates are nilext. (b) shows the distribution of percentage of reads within T_f; a bar for x%-y% shows the percentage of clusters where x%-y% of reads access objects updated within T_f. We consider T_f=1s, 50ms.*

interface (to not return errors) can enable a replication protocol to realize higher performance.

We performed a similar analysis on the IBM-COS traces across 35 storage clusters with at least 10% writes (out of 98 in total) [24]. COS supports three kinds of updates: put, copy, and delete. While put and copy are nilext, delete is not; it returns an error if the object does not exist. In about 65% clusters, more than half of the updates are nilext; these operations can be completed quickly. Again, if the semantics of delete can be modified, all updates can be made faster.

We next analyze how often reads may incur overhead. A read will incur overhead if there are unfinalized updates to the object being read. Let T_f be the time taken to finalize updates. We thus measure the time interval between a read to an object and the prior write to the same object, and calculate the percentage of reads for which this interval is less than T_f. We use the IBM-COS traces for this analysis because the Twemcache traces do not have millisecond-level timestamps.

Figure 3(b) shows the distribution of percentage of reads that access items updated within T_f. We first consider T_f to be 1s. Even with such an unrealistically high T_f, in 66% of clusters, only less than 5% of reads access objects modified within 1s. We next consider a more realistic T_f of 50ms. T_f=50ms is realistic (but still conservative) because these traces are from a setting where replicas are in different zones of the same geographical region, and inter-zone latencies are ~2 ms [38]. With T_f=50 ms, in 85% of clusters, less than 5% of reads access objects modified within 50 ms; thus, only a small fraction of reads in a nilext-aware protocol may incur overhead in practice. Further, not all such reads will incur overhead due to prior reads to unfinalized updates and non-nilext updates that would force synchronous ordering.

3.4 Comparison to Other Approaches

While nilext-aware replication defers ordering, prior work has built solutions to efficient ordering. The nilext-aware approach offers advantages over such prior solutions. While we focus on consensus-based approaches here, other ways to construct replicated storage systems exist; we discuss how exploiting nil-externality applies to them as well.

3.4.1 Efficient Ordering in Consensus.

Prior approaches to efficient ordering broadly fall into three categories.

Network Ordering. This approach enforces ordering in the network [21, 50]: the network consistently orders requests across replicas in one RTT, improving performance. In contrast, a nilext-aware protocol does not require a specialized network and thus applies to geo-replication as well.

Speculative Execution. This approach employs speculative execution to reduce ordering cost [42, 67]. Replicas speculatively execute requests before agreeing on the order. Clients then compare responses from different replicas to detect inconsistencies and replicas rollback their state upon divergence. Replicas can thus be in an inconsistent state before the end application is acknowledged. However, when end application is notified, the system ensures that the requests have been executed in the correct order. In contrast, the nature of nilext interfaces allows one to defer ordering and execution even after the application is notified of completion; only durability must be ensured before notifying. Ordering and execution are performed only when the effects are externalized by *later* operations. Also, a nilext-aware protocol does not require replicas to do rollbacks, reducing complexity.

Exploiting Commutativity. This approach (used in Generalized Paxos [45], EPaxos [58]) realizes that ordering is not needed when updates commute. Both commutative and nilext-aware protocols incur overhead when reads access unfinalized updates. However, as we show (§5.7), commutative protocols can be expensive when updates conflict and when operations do not commute. Nilext-aware replication, in contrast, always completes nilext updates in one RTT. Finally, nil-externality and commutativity are not at odds: a nilext-aware protocol can exploit commutativity to commit non-nilext writes faster (§5.7).

3.4.2 Other Approaches to Replicated Storage.

Shared registers [4], primary-backup [9], and chain replication [76] offer other ways to building replicated storage. Storage systems that support only reads and writes can be built using registers which are not subject to FLP impossibility [4]. However, shared registers cannot readily enable RMWs [2, 10], a common requirement in modern storage APIs. Starting with state machines as the base offers more flexibility and exploiting nil-externality when possible leads to high performance. Gryff [10] combines registers (for reads and writes) and consensus (for RMWs); however, Gryff's writes take 2 RTTs. Primary-backup, chain replication, and other approaches [19] support a richer API. However, primary-backup also incurs 2 RTTs for updates [51, 64]; similarly, updates in chain replication also incur many message delays. The idea of exploiting nil-externality can be used to hide the ordering cost in these approaches as well; we leave this extension as an avenue for future work.

Summary. Unlike existing approaches, nilext-aware replication takes advantage of nil-externality of storage interfaces.

Client interface	Upcalls into storage system
`InvokeNilext(req)`	`MakeDurable`
sent to all replicas	add a nilext update to durability log
wait for ack from supermajority	`Read`
(including one from the leader)	read item; returns <need_sync, data>
`InvokeNonNilext(req)`	`Apply`
`InvokeRead(req)`	apply request to state; optionally return result
sent only to the leader	`GetDurabilityLogEntries`
wait for result from the leader	used in background ordering and view-change

Figure 4. Client Interface and Upcalls. *The figure shows the client interface and the upcalls the replication layer makes into the storage system.*

It should perform well in practice: nilext updates contribute to a large fraction of writes and reads do not often access recent updates. This approach offers advantages over existing efficient ordering mechanisms: it requires no network support; it can defer execution beyond request completion and does not require rollbacks; it offers advantages over and combines well with exploiting commutativity.

4 SKYROS Design and Implementation

We now describe the design of SKYROS. We first provide an overview (§4.1), describe normal operation (§4.2 – §4.5), and explain recovery and view change (§4.6). We then show the correctness of SKYROS (§4.7). We finally discuss practical issues we addressed in SKYROS (§4.8).

4.1 Overview

We use VR (or multi-paxos) as our baseline to highlight the differences in SKYROS. VR tolerates up to f failures in a system with $2f + 1$ replicas. It is leader-based and makes progress in a sequence of views; in each view, a single replica serves as the leader. VR implementations offer linearizability [36]: operations are executed in real-time order, and each operation sees the effect of ones that completed before it. SKYROS preserves all these properties: it provides the same availability, is leader-based, and offers linearizability.

In VR, the leader establishes an order by sending a *prepare* and waiting for *prepare-ok* from f followers. The leader then does an *Apply* upcall into the storage system to execute the operation. SKYROS changes this step in an important way: while SKYROS makes updates immediately durable, it defers ordering and executing them until their effects are externalized. To enable this, SKYROS augments the interface between the storage system and the replication layer with additional upcalls (as shown in Figure 4). During normal operation, SKYROS processes different requests as follows:

- Clients submit nilext updates to all replicas using *InvokeNilext*. Since nil-externality is a static property (it does not depend upon the system state), clients can decide which requests are nilext and invoke the appropriate call. Upon receiving a nilext update, replicas invoke the *MakeDurable* upcall to make the operation durable (§4.2).
- Although nilext updates are not immediately finalized, they must be executed in the same real-time order across replicas. The leader gets the replicas to agree upon an order and the replicas apply the updates in the background (§4.3).
- Clients send read requests to the leader via *InvokeRead*. When a read arrives, the leader does a *Read* upcall. If all updates that the read depends upon are already applied, the read is served quickly; otherwise, the leader orders and executes updates before serving the read (§4.4).
- Clients send non-nilext updates to the leader via *InvokeNonNilext*; such updates are immediately finalized (§4.5).

4.2 Nilext Updates

Clients send nilext updates directly to all replicas including the leader to complete them in one RTT. Each request is uniquely identified by a sequence number, a combination of client-id and request number. Similar to VR, only replicas in the *normal* state reply to requests and duplicate requests are filtered. A replica stores the update by invoking *MakeDurable*. SKYROS replicas store these durable (but not yet ordered or applied) updates in a separate *durability log*; each replica thus has two logs: the usual consensus log and the durability log. Once a replica stores the update in the durability log, it responds directly to the client; the replica adds its current view number in the response. For a nilext update, clients wait for a *supermajority* of $f + \lceil f/2 \rceil + 1$ acknowledgments in the same view including one from the leader of the view. Figure 5(a)(i) shows how a nilext update a is completed.

Note that an update need not be added in the same position in the durability logs across replicas. For example, in Figure 5(b)(i), b is considered completed although its position is different across durability logs. Then, why do SKYROS replicas use a durability *log* instead of a *set*? Using an unordered set precludes the system from reconstructing the required ordering between updates upon failures. For example, in Figure 5(b)(i) and (b)(ii), b follows a in real time (i.e., a completed before b started) and thus must be applied to the storage system only after a. A log captures the order in which the replicas receive the requests; SKYROS uses these logs to determine the ordering of requests upon failures.

Why is a simple majority ($f + 1$) insufficient? Consider an update b that follows another update a in real-time. Let's suppose for a moment that we use a simple majority. A possible state then is $< D_1 : ab, D_2 : ab, D_3 : ab, D_4 : ba, D_5 : ba >$, where D_i is the durability log of replica S_i. This state is possible because a client could consider a to be completed once it receives acknowledgment from S_1, S_2, and S_3. Then, b starts and is stored on all durability logs and so is considered completed. a now arrives late at S_4 and S_5. Assume the current leader (S_1) crashes. Now, we have four replicas whose logs are $< D_2 : ab, D_3 : ab, D_4 : ba, D_5 : ba >$. With these logs, one cannot determine the correct order. A supermajority quorum avoids this situation. Writing to a supermajority ensures that a majority within any available majority is guaranteed to have the requests in the correct order in their durability logs. We later show how by writing to a supermajority, SKYROS recovers the correct ordering upon failures (§4.6, §4.7).

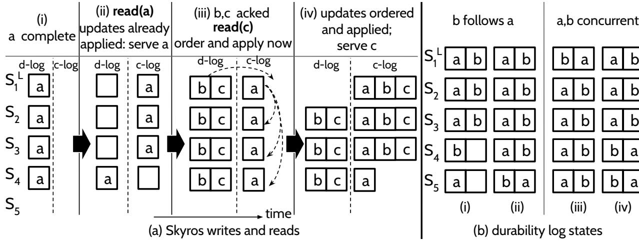


Figure 5. SKYROS Writes and Reads, and Durability Log States. *(a) shows how* Skyros *processes nilext writes and reads; d-log: durability, c-log: consensus log, L: leader; f=2 and supermajority=4. (b) shows the possible durability logs for two completed nilext operations a and b. In (i) and (ii), b follows a in real time, whereas in (iii) and (iv), they are concurrent.*

During normal operation, the leader's durability log is guaranteed to have the updates in the correct order. This is because a response from the leader is necessary for a request to complete. Thus, if an update b follows another update a in real-time, then the leader's durability log is guaranteed to have a before b (while some replicas may contain them in a different order as in Figure 5(b)(ii)). This guarantee ensures that when clients read from the leader, they see the writes in the correct order. The leader uses this property to ensure that operations are finalized to the consensus log in the correct order. If a and b are concurrent, they can appear in the leader's log in any order as in Figure 5(b)(iii) and (b)(iv).

4.3 Background Ordering and Execution

While nilext updates not are immediately ordered, they must be ultimately executed in the same real-time order across replicas. The leader is guaranteed to have all completed updates in its durability log in real-time order. Periodically, the leader takes an update from its durability log (via the *GetDurabilityLogEntries* upcall), adds it to the consensus log, and initiates the usual ordering protocol. Once f followers respond after adding the request to their consensus logs, the leader applies the update and removes it from its durability log. At this point, the request is finalized. As in VR, the leader sends a *commit* for the finalized request; the followers apply the update and then remove it from their durability logs. Note that this step is the same as in VR; once $f + 1$ nodes agree on the order, at least one node in any majority will have requests in the correct order in its consensus log.

The leader employs batching for the background work; it adds many requests to its consensus log and sends one *prepare* for the batch. Once f followers respond, it applies batch and removes it from the durability log.

4.4 Reads

Clients read only at the leader in SKYROS (like in many linearizable systems). When a read arrives, the leader does a *Read* upcall. The storage system then performs an ordering and execution check: it consults the durability log to check if there are any pending updates that this read depends upon. For example, a key-value store would check if there is a pending put or merge to the key being read. Note that this check is system-specific, which led to our design rationale of maintaining the durability log within the storage system, giving it visibility in to the pending updates to perform the check. The storage system maintains an efficient index (such as a hash table) to quickly lookup the log.

If there are no pending updates, the storage system populates the response by reading the state, sets the *need_sync* bit to 0, and returns the read value to the replication layer. The leader then returns the response to the client, completing the read in one RTT (e.g., read-a in Figure 5(a)(ii)).

Conversely, if there are pending updates, the storage system sets the *need_sync* bit. In that case, the leader synchronously adds all requests from the durability log to the consensus log to order and execute them (e.g., read-c in Figure 5(a)(iii)). Once f followers respond, the leader applies all the updates and then serves the read. Fortunately, the periodic background finalization reduces the number of requests that must be synchronously ordered and executed during such reads.

4.5 Non-nilext Updates

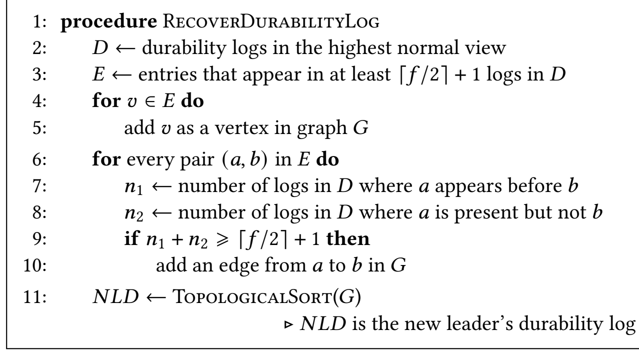
If an update externalizes state, then it must be immediately ordered and executed. Clients send such non-nilext updates only to the leader. The leader first adds all prior requests in the durability log to the consensus log; it then adds the non-nilext update to the end of the consensus log and then sends a *prepare* for all the added requests. Once f followers respond, the leader applies the non-nilext update (after applying all prior requests) and returns the result to the client.

4.6 Replica Recovery and View Changes

So far, we have described only the failure-free operation. We now discuss how SKYROS handles failures.

Replica Recovery. Similar to VR, SKYROS does not write log entries synchronously to disk (although it maintains view information on disk). Thus, when a replica recovers from a crash, it needs to recover its log. In VR, the replica marks its status as *recovering*, sends a RECOVERY message, and waits for a RECOVERYRESPONSE from at least $f + 1$ replicas, including one from the leader of the latest view it sees in these responses [52]. Then, it sets its log as the one in the leader's response. The replica then sets its status to *normal*. Recovery in SKYROS is very similar with one change: the replicas also send their durability logs in RECOVERYRESPONSE and a replica sets its durability log as the one sent by the leader. This step is safe because the leader's durability log contains all completed nilext updates in the correct order.

View Changes. In VR, when the leader of the current view fails, the replicas change their status from *normal* to *view-change* and run a view-change protocol. The new leader must recover all the committed operations in the consensus log before the system can accept requests. The new leader does this by waiting for f other replicas to send a DOVIEWCHANGE message [52]. In this message, a replica includes its view number, its log, and the last view number in which its status was *normal*. The leader then recovers the log by taking the



```
 1: procedure RecoverDurabilityLog
 2:     D ← durability logs in the highest normal view
 3:     E ← entries that appear in at least ⌈f/2⌉ + 1 logs in D
 4:     for v ∈ E do
 5:         add v as a vertex in graph G
 6:     for every pair (a, b) in E do
 7:         n1 ← number of logs in D where a appears before b
 8:         n2 ← number of logs in D where a is present but not b
 9:         if n1 + n2 ⩾ ⌈f/2⌉ + 1 then
10:             add an edge from a to b in G
11:     NLD ← TopologicalSort(G)
                        ▷ NLD is the new leader's durability log
```

Figure 6. RecoverDurabilityLog. *The figure shows the procedure to recover the durability log at the leader during a view change.*

most up-to-date[†] one among the $f+1$ logs (including its own). The leader then sends a StartView message to the replicas in which it includes its log; the leader sets its status as *normal*. The replicas set their consensus log as the one sent by the leader after which they set their status as *normal*. Skyros uses exactly the same procedure to recover operations that have been finalized (i.e., operations in the consensus log). Thus, finalized operations are safely recovered as in VR.

In Skyros, the leader must additionally recover the durability log. The previous leader's durability log would have contained all completed operations. Further, the previous leader's durability log would have contained the completed operations in the correct real-time order, i.e., if an operation a had completed before b, then a would have appeared before b. These same guarantees must be preserved in the new leader's durability log during a view change.

Skyros replicas send their durability logs as well in the DoViewChange message. However, it is unsafe for the new leader to take one log in the responses as its durability log; a single log may not contain all completed operations. Consider three completed updates a, b, and c, and let the durability logs be $< D_1 : abc, D_2 : ac, D_3 : abc, D_4 : ab, D_5 : bc >$. If S_2, S_4, and S_5 participate in a view change, no single log would contain all completed operations. Even if a single durability log has all completed operations, it may not contain them in the correct real-time order. Consider a completes before b starts, and c is incomplete and let the durability logs be $< D_1 : ab, D_2 : ab, D_3 : bac, D_4 : ab, D_5 : ab >$. If S_2, S_3, and S_4 participate in a view change, although D_3 contains all completed operations, taking D_3 as the leader's log will violate linearizability because b appears before a in D_3.

To correctly recover the durability log, a Skyros leader uses the RecoverDurabilityLog procedure (Figure 6). We use Figure 7 to illustrate how this procedure works. In this example, f=2; operations a, b, and c completed, while d did not. a and b were concurrent with each other, and c started after a and b completed. Thus, the new leader must recover a, b,

[†]i.e., the log from a replica with the largest normal view; if many replicas have the same normal view, the largest log among them is chosen.

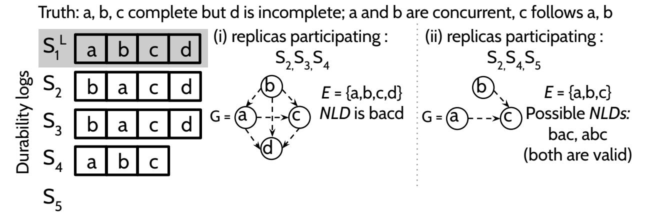


Figure 7. RecoverDurabilityLog Example. *The figure shows how RecoverDurabilityLog works. S_1, the leader of the previous view view-1, has failed; this is a view-change for view-2 for which S_2 is the leader.*

and c; also, c must appear after a and b in the recovered log.

The system must make progress with f failures; thus, the procedure must correctly recover the durability log with $f+1$ replicas participating in a view change. As in VR, upon receiving $f+1$ DoViewChange messages, the leader first finds the highest normal view from the responses and considers all durability logs in that view; we denote this set of logs as D (line 2). For example, in Figure 7(i), S_2, S_3, and S_4 participate in the view change and the last normal view of all replicas is 1. Therefore, D_2, D_3, and D_4 are part of D. To recover completed operations, the leader then checks which operations appear in at least $\lceil f/2 \rceil + 1$ logs in D. Such operations are the ones that the leader will recover as part of the new durability log; we denote this set as E (line 3). For example, in Figure 7(i), a, b, c, and d are part of E (as they all appear in $\geqslant 2$ logs); similarly, in Figure 7(ii), a, b, and c are part of E.

The above steps give the operations that form the durability log, but not the real-time order among them. To determine the order, the leader considers every pair of operations $< x, y >$ in E, and counts the number of logs where x appears before y or x appears but y does not. If this count is at least $\lceil f/2 \rceil + 1$, then the leader determines that y follows x in real time. In Figure 7(ii), a appears before c on $\geqslant 2$ logs and so the leader determines that c follows a. In contrast, a does not appear before b (or vice versa) in $\geqslant 2$ logs and thus are concurrent. Thus, this step gives only a partial order.

The leader constructs the total order as follows. It first adds all operations in E as vertices in a graph, G (lines 4–5). Then, for every pair of vertices $< a, b >$ in G, an edge is added between a and b if on at least $\lceil f/2 \rceil + 1$ logs, either a appears before b, or a is present but not b (lines 6–10). G is a DAG whose edges capture the real-time order between operations. To arrive at the total order, the leader topologically sorts G (line 11) and uses the result as its durability log (NLD). In Figure 7(ii), both bac and abc are valid total orders.

The leader then appends the operations from the durability log to the consensus log; duplicate operations are filtered using sequence numbers. Then, the leader sets its status as *normal*. The leader then sends the consensus log in the StartView message to the replicas (similar to VR). The followers, on receiving StartView, replace their consensus logs with the one sent by the leader and set their status to *normal*. The system is now available to accept new requests.

4.7 Correctness

We now show that Skyros is correct. Two correctness conditions must be met. **C1**: all completed and finalized operations remain durable, **C2**: all operations are applied in the linearizable order and an operation finalized to a position survives in the same position. The proof sketch is as follows.

C1. Ensuring durability when the leader is alive is straightforward; a failed replica can recover its state from the leader. Durability must also be ensured during view changes; the new leader must recover all finalized and completed operations. Finalized operations are part of at least $f+1$ consensus logs. Thus, at least one among the $f+1$ replicas participating in the view change is guaranteed to have the finalized operations and thus will be recovered (this is similar to VR).

Next we show that completed operations that have not been finalized are recovered. Let v be the view for which a view change is happening and the highest normal view be v'. We first establish that any operation that completed in v' will be recovered in v. Operations are written to $f+\lceil f/2 \rceil+1$ durability logs before they are considered completed and are not removed from the durability logs before they are finalized. Therefore, among the $f+1$ replicas participating in the view change for v, a completed operation in v' will be present in at least $\lceil f/2 \rceil + 1$ durability logs. Because the new leader checks which operations are present in at least $\lceil f/2 \rceil + 1$ logs (line 2 in Figure 6), operations completed in v' that are not finalized will be recovered as part of the new leader's durability log.

We next show that operations that were completed in an earlier view v'' will also survive into v. During the view change for v', the leader of v' would have recovered the operations completed in v'' as part of its durability log (by the same argument above). Before the view change for v' completed, the leader of v' would have added these operations from its durability log to the consensus log. Any node in the *normal* status in view v' would thus have these operations in its consensus log. Consensus-log recovery would ensure these operations remain durable in successive views including v.

C2. During normal operation, the leader's durability log reflects the real-time order. The leader adds operations to its consensus log only in order from its durability log. Before an (non-nilext) operation is directly added to the consensus log, all prior operations in the durability log are appended to the consensus log as well. Thus, all operations in the consensus log reflect the linearizable order. Reads are served by the leader which is guaranteed to have all acknowledged operations; thus, any read to an object will include the effect of all previous operations. This is because the leader ensures that any pending updates that the read depends upon are applied in a linearizable order before the read is served.

The correct order must also be maintained during view changes. Similar to VR, the order established among the finalized operations (in the consensus log) survives across views; any operation committed to the consensus log will survive in the same position.

Next, we show that the linearizable order of completed-but-not-finalized operations is preserved. As before, we need to consider only operations that were completed but not yet finalized in v'; remaining operations will be recovered as part of the consensus log. We now show that for any two completed operations x and y, if y follows x in real time, then x will appear before y in the new leader's recovered durability log. Let G be a graph containing all completed operations as its vertices. Assume that for any pair of operations $< x, y >$, a directed edge from x to y is correctly added to G if y follows x in real time (*A1*). Next assume that G is acyclic (*A2*). If *A1* and *A2* hold, then a topological sort of G ensures that x appears before y in the result of the topological sort. We show that *A1* and *A2* are ensured by Skyros.

A1: Consider two completed operations a and b and that b follows a in real time. Since a completed before b, when b starts, a must have already been present on at least $f + \lceil f/2 \rceil + 1$ durability logs; let this set of logs be DL. Now, for each log dl in DL, if b is written to dl, then b would appear after a in dl. If b is not written to dl, then a would appear in dl but not b. Thus, a appears before b or a is present but not b on at least $f + \lceil f/2 \rceil + 1$ durability logs. Consequently, among the $f + 1$ replicas participating in view change, on at least $\lceil f/2 \rceil + 1$ logs, a appears before b or a is present but not b. Because the leader adds an edge from a to b when this condition is true (lines 7–9 in Figure 6) and because it considers all pairs, *A1* is ensured. *A2:* Since $\lceil f/2 \rceil + 1$ is a majority of $f + 1$, an opposite edge from b to a would not be added to G. Since all pairs are considered, G is acyclic.

A completed operation is assigned a position only when it is finalized. Since Skyros adds an operation from the durability log to the consensus only if it is already not present in the consensus log, a completed operation is finalized only once, after which it survives in the finalized position.

Model Checking. We have modeled the request-processing and view-change protocols in Skyros, and model checked them. We explored over 2M states, in which the above correctness conditions were met. Upon modifying the specification in subtle but wrong ways, our model checker finds safety violations. For example, in the RecoverDurabilityLog procedure, an edge is added from a to b when a appears before b in $\lceil f/2 \rceil + 1$ logs; if this threshold is increased, then a required edge will not be added, leading to a linearizability violation that the checker correctly flags; decreasing the threshold makes G cyclic, triggering a violation. Similarly, the checker finds a safety violation if durability-log entries are not added to consensus log before sending StartView.

4.8 Practical Issues and Solutions

We now describe a few practical problems we handled in Skyros. We also discuss possible optimizations.

Space and Catastrophic Errors. Because nilext updates are not immediately executed, certain errors cannot be detected. For instance, an operation can complete but may fail later when applied to the storage system due to insufficient space. A protocol that immediately executes operations, in theory, could propagate such errors to clients. However, such space errors can be avoided in practice by using space watermarks that the replication layer has visibility into; once a threshold is hit, the replication layer can throttle updates while the storage system reclaims space. One cannot, however, anticipate catastrophic memory or disk failures. Fortunately, this is not a major concern in practice. Given the inherent redundancy, a SKYROS replica transforms such errors into a crash failure; it is unlikely that all replicas will encounter the same error. Note that these are errors that are not part of the nilext interface contract. SKYROS checks for all validation errors in the *MakeDurable* upcall.

Determining Nil-externality. While it is straightforward in many cases to determine whether or not an interface is nilext, occasionally it is not. For instance, a database update may invoke a trigger which can externalize state. However, when unsure, clients can safely choose to say that an interface is non-nilext, forgoing some performance for safety.

Replica-group Configuration and Slow Path. In our implementation, clients know the addresses of replicas from a configuration value. During normal operation, SKYROS clients contact all replicas in the group and wait for a supermajority responses to complete nilext writes. If the system is operating with a bare majority, then writes cannot succeed, affecting availability. SKYROS handles this situation using a slow path: after a handful of retries, clients mark requests to be non-nilext and send it to the leader. These requests are acknowledged after they are committed to a majority consensus logs, allowing clients to make progress.

Possible Optimizations. In SKYROS, requests are initially stored in the durability log. The leader later adds the requests to its consensus log and replicates the consensus log. Our current implementation sends the requests in their entirety during background replication. This is unnecessary in most cases because the replicas already contain the request in their durability logs. A more efficient way would be to send only the ordering information (i.e., the sequence numbers). Second, locally, a copy between the durability log and the consensus log can be avoided if the entries are stored in a separate location and the log slots point to the entries. Finally, SKYROS allows reads only at the leader; the burden on the leader can be alleviated by using techniques such as quorum reads [12] without impacting linearizability. We leave these optimizations as an avenue for future work.

5 Evaluation

To evaluate SKYROS, we ask the following questions:

- How does SKYROS perform compared to standard replication protocols on nilext-only workloads? (§5.1)
- How does SKYROS perform on mixed workloads? (§5.2)
- How do read-latest percentages affect performance? (§5.3)
- Does the supermajority requirement in SKYROS impact performance with many replicas? (§5.4)
- How does SKYROS perform on YCSB workloads? (§5.5)
- Does replicated RocksDB benefit from SKYROS? (§5.6)
- Does SKYROS offer benefit over commutative protocols? Is nil-externality compatible with commutativity? (§5.7)

Setup. We run our experiments on five replicas; thus, f=2 and *supermajority*=4. Each replica runs on a m5zn bare-metal instance [5] in AWS (US-East). Numbers reported are the average over three runs. Our baseline is VR/multi-paxos which implements batching to improve throughput (denoted as Paxos). SKYROS also uses batching for background work. Most of our experiments use a hash-table-based key-value store; however, we also show cases with RocksDB.

5.1 Microbenchmark: Nilext-only Workload

We first compare the performance for a nilext-only workload. Figure 8(a) plots the average latency against the throughput when varying the number of clients. We also compare to a no-batch Paxos variant in this experiment. In all further experiments, we compare only against Paxos with batching.

We make three observations from the figure. First, SKYROS and Paxos offer ~3× higher throughput than the Paxos no-batch variant. Second, with a small number of clients, SKYROS offers ~2× better latency and throughput than Paxos with batching. Batching across many clients improves the throughput of Paxos. However, this affects latency: at about 100 KOps/s, SKYROS offers 3.1× lower latency than Paxos.

5.2 Microbenchmark: Mixed Workloads

We next consider mixed workloads. We use 10 clients.

Nilext and non-nilext writes. Figure 8(b)(i) shows the result for a workload with a mix of nilext and non-nilext writes. With low non-nilext fractions, SKYROS offers 2× higher throughput because most writes complete in 1 RTT. As the non-nilext fraction increases, the benefits of SKYROS reduces. However, even in the worst case where all writes are non-nilext, SKYROS does not perform worse than Paxos. As noted earlier, in many deployments, the fraction of non-nilext writes is low and thus SKYROS would offer benefit; for example, with 10% non-nilext writes, SKYROS offers ~78% higher throughput.

Nilext and reads. We next consider a workload with nilext writes and reads. In SKYROS, if a read accesses a key for which there are unfinalized updates, the read will incur 2 RTTs. We thus consider two request distributions: uniform and zipfian. We vary the percentage of writes (W) and show the mean and p99 latency in Figure 8(b)(ii). In the uniform case, operations do not often access the same keys and thus reads rarely incur 2 RTTs. With a low W, SKYROS offers only little benefit with mean latency (e.g., 10% lower mean latency with 10% writes). However, SKYROS reduces p99 latency by 80% because writes are faster and reads rarely incur 2 RTTs.

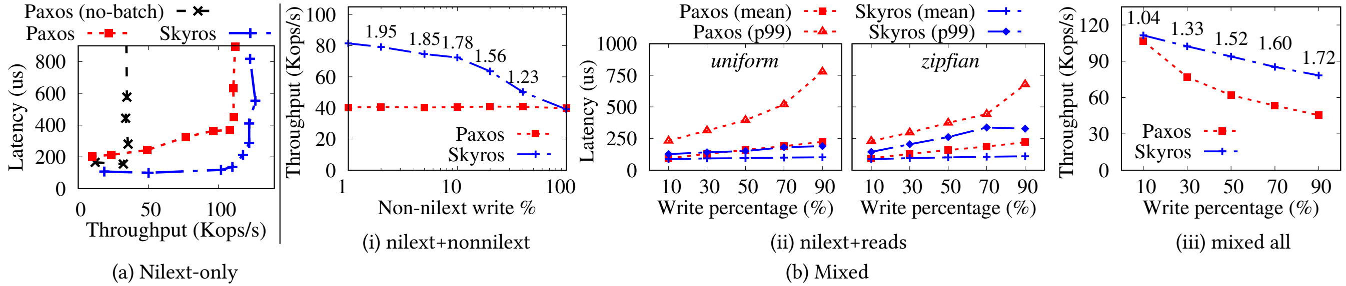


Figure 8. Microbenchmark: Different Workload Mixes. *(a) compares the performance of* Skyros *to Paxos for a nilext-only workload. (b) shows the performance under three different mixed workloads (nilext+nonnilext, nilext+reads, and nilext+nonnilext+reads).*

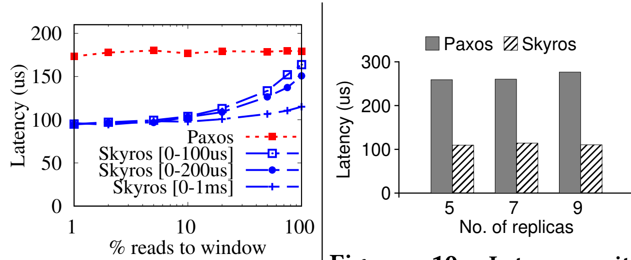


Figure 9. Read-latest. *The figure shows the performance of* Skyros *with varying read-latest percentages.*

Figure 10. Latency with Many Replicas. *The figure compares the average latency of* Skyros *for different cluster sizes.*

With a high W (90%), SKYROS offers significant benefit: it reduces mean latency by 2.2× and p99 latency by 4.1×.

In the zipfian case, some keys are more popular than others. Therefore, reads may often access keys recently modified by writes. Thus, as shown, p99 latency in SKYROS for zipfian increases compared to the uniform case. However, not all reads incur 2 RTTs because of background finalization and prior reads that force synchronous ordering. Thus, although the improvements decrease compared to the uniform case, SKYROS still offers significant benefit over Paxos (e.g., at W = 90%, mean and p99 latencies in SKYROS are 2× lower).

Writes and reads. We next run a mixed workload with all three kinds of operations. We vary the write percentage (W) and fix the non-nilext fraction to be 10% of W. As shown in Figure 8(b)(iii), with a small fraction of writes, SKYROS offers little benefit over Paxos because reads take 1 RTT in both systems. With a higher W, SKYROS offers higher performance; for example, with W=90% (9% non-nilext), SKYROS offers 1.72× higher throughput.

5.3 Microbenchmark: Read Latest

If many reads access recently modified items, then SKYROS would incur overhead on reads. To show this, we run a workload with 50% nilext writes and 50% reads with 10 clients. We vary the amount of reads that access items that were updated within three different windows [0-100] us (roughly 1 RTT on our testbed), [0-200] us (roughly 2 RTTs), and [0-1] ms (a large window), and measure the average request latency.

Figure 9 shows the result. Intuitively, if no or few reads access recently modified items, then performance of SKYROS would not be affected by reads taking 2 RTTs (leftmost point of the graph). SKYROS offers ~70% lower latency than Paxos. As we increase percentage of reads accessing items updated in the window, more reads incur 2 RTTs and thus the average latency increases. Moreover, latency increases more steeply for smaller windows; for example, when all reads go to items updated in the last 100 us, many reads (~68%) incur 2 RTTs. Again, not all reads incur 2 RTTs because of background finalization and prior reads to the items that force synchronous ordering. In common workloads, where reads do not often access recently written items, SKYROS offers advantages. For example, with 10% reads accessing items updated in last 100 us, SKYROS offers 70% lower latency.

5.4 Microbenchmark: Latency with Many Replicas

In prior experiments, we use five replicas and thus clients wait for four responses. With larger clusters, SKYROS clients must wait for many responses (e.g., seven responses with nine replicas), potentially increasing latency. To examine this, we conduct an experiment with seven and nine replicas and measure the latencies for a nilext-only workload with 10 clients. As shown in Figure 10, the additional responses do not add much to the latencies; latencies in the seven and nine-node configurations are similar to that of the five-replica case (about 110μs) and is about 2× lower than Paxos.

Microbenchmark Summary. SKYROS offers benefit under many workloads with different request ratios and distributions. Even when pushed to extreme cases (e.g., all non-nilext or all reads access recent writes), SKYROS does not perform worse than Paxos. Under realistic workloads, SKYROS offers higher throughput, and lower mean and tail latencies.

5.5 YCSB Macrobenchmark

We next analyze performance under six ycsb [16] workloads: Load (write-only), A (50% w, 50% r), B (5% w, 95% r), C (read-only), D (5% w, 95% r), and F (50% rmw, 50% r). Figure 11(a) shows the result for 10 clients. For write-heavy workloads (load, A, and F), SKYROS improves throughput by 1.43× to 2.29×. SKYROS offers similar performance for the read-only workload. For read-heavy workloads (B and D), SKYROS offers little benefit; only 5% of operations can be made faster.

To understand the effect of reads that trigger synchronous ordering, we examine the read-latency distributions (Figure 11(b) and (d)). In both ycsb-a and ycsb-b, most reads

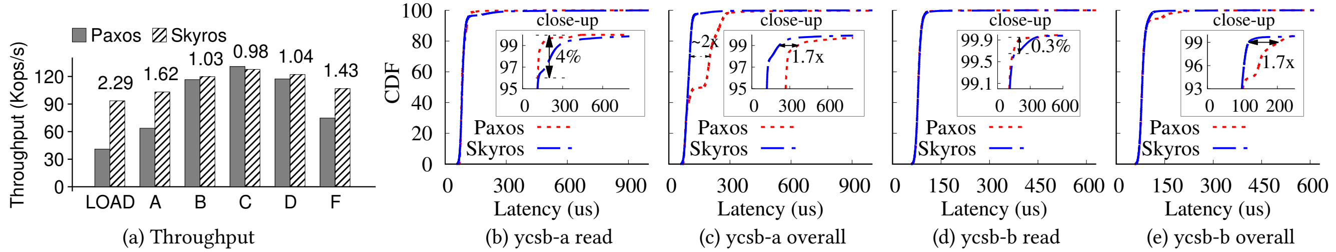


Figure 11. YCSB Performance. *(a) show the throughput for all ycsb workloads; (b) and (d) show the read-latency distribution for ycsb-a and ycsb-b, respectively; (c) and (e) show the operation-latency distribution for the same workloads.*

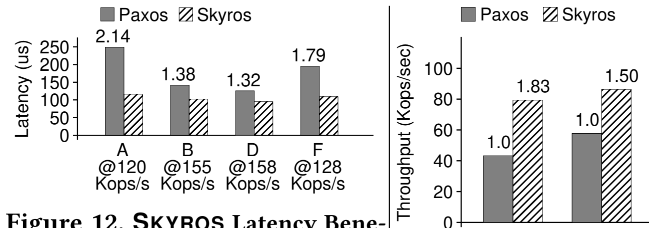


Figure 12. SKYROS Latency Benefits. *The figure compares the average latency at maximum throughput for mixed YCSB workloads. The number below each bar shows the throughput for the workload.*

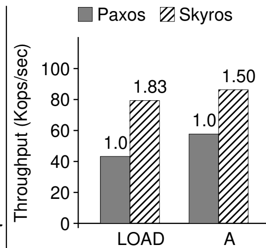


Figure 13. RocksDB. *The figure shows performance in RocksDB.*

complete in 1 RTT, while some incur overhead. However, this fraction is very small (e.g., 4% in ycsb-a and 0.3% in ycsb-b; we see similar fractions for other workloads too). However, the slow reads do not affect the overall p99 latency. In fact, examining the distribution of operation (both read and write) latencies shows that SKYROS reduces the overall p99 latency. This reduction arises because the tail in the overall workload includes expensive writes in Paxos, which SKYROS makes faster. As a result, SKYROS reduces overall p99 latency by 1.7× in ycsb-a and ycsb-b as shown in Figure 11(c) and (e).

Latency Benefits. For a fixed number of clients as in the previous experiment, SKYROS offers higher throughput than Paxos. This is because, in baseline Paxos, the leader waits for requests to be ordered in 2-RTTs. While SKYROS defers this ordering work, it does not avoid it. However, by moving the ordering-wait in Paxos to the background, SKYROS is able to use the otherwise idle CPU cycles to accept more requests; this enables SKYROS to achieve higher throughput.

Paxos, with batching across many clients, can achieve high throughput levels (similar to SKYROS). However, at such high throughput, SKYROS offers significant latency benefits. To illustrate this, we measure the average latency at the maximum throughput obtained by Paxos for write-heavy (ycsb-a,f) and read-heavy (ycsb-b,d) workloads. As shown in Figure 12, SKYROS offers 1.32×–2.14× lower latencies than Paxos for the same throughput.

5.6 Replicated RocksDB: Paxos vs. SKYROS

We have also integrated RocksDB with SKYROS. We built a wrapper around RocksDB in which we implemented the upcalls. Figure 13 compares the performance under two workloads when using SKYROS and Paxos to replicate RocksDB. As before, SKYROS offers notable improvements.

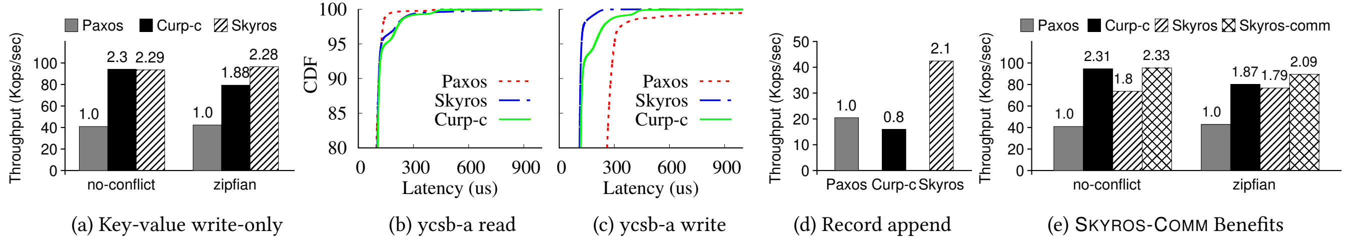
5.7 Comparison to Commutative Protocols

We now compare SKYROS to commutative protocols. We compare against Curp [64], a recent protocol that improves over prior commutative protocols. Curp targets primary-backup, but sketches the protocol for consensus [64, §Appendix-B.2]. In this protocol, a client sends an update u to all replicas; each replica adds u to a witness component if u commutes with prior operations in the witness. The leader adds u to the log, executes u speculatively, and returns a response. Clients wait for a supermajority responses (including the leader's result). If the leader detects a conflict, it initiates a *sync*, finishing the operation in 2 RTT. If a conflict arises at the followers, the client detects that and informs the leader to initiate a *sync*; such requests take 3 RTTs. Reads are sent only to the leader and thus would incur only 2 RTT upon conflicts. We implement this protocol and call our implementation Curp-c.

5.7.1 Benefits over Commutative Protocols.

We first compare SKYROS and Curp-c under a write-only key-value workload (only set). Figure 14(a) shows the result. In the no-conflict case (no two writes access the same key), Curp-c and SKYROS perform similarly and are 2× faster than Paxos. In Curp-c, all requests take 1 RTT because no request conflicts with another. In SKYROS, all operations are nilext and so complete in 1 RTT. However, for a zipfian workload ($\theta = 0.99$, the default in YCSB), Curp-c's performance drops due to conflicts, while SKYROS maintains the high performance. In this case, SKYROS offers 2.7× lower p99 latency than Curp-c.

We next run ycsb-a (50%w, 50%r). As shown in Figure 14(b), Paxos reads take 1 RTT. In SKYROS, a small fraction of reads take 2 RTTs. A similar fraction of reads in Curp-c also conflict with prior writes and thus incur 2 RTTs. As shown in Figure 14(c), nilext writes in SKYROS can always complete in 1 RTT. In contrast, in Curp-c, writes conflict with prior writes and thus sometimes incur 2 or 3 RTTs. As a result, SKYROS offers 34% lower p99 latency. We observe that write-write conflicts in Curp-c lead to 50% more slow-path operations than read-write conflicts in SKYROS and Curp-c. A write-write conflict can arise due to unsynced operations on any replica, whereas a read-write conflict can occur only at the leader. Further, the followers' knowledge of synced operations is behind the leader by a message delay, increasing the conflict window at the followers.



(a) Key-value write-only (b) ycsb-a read (c) ycsb-a write (d) Record append (e) Skyros-Comm Benefits

Figure 14. Comparison to Commutativity. *(a) shows the write throughput in kv-store. (b) and (c) show the latencies for ycsb-a. (d) compares record-append throughput. (e) shows the kv-store throughput for a nilext + non-nilext workload.*

Exploiting nil-externality offers benefit over commutativity when operations do not commute. To show this, we built a file store that supports GFS-style record appends [32]. The record-append interface is not commutative: records must be appended in the same order across replicas. However, it is nilext: it just returns a *success*. Figure 14(d) shows the result when four clients append records to a file. Because every operation conflicts, Curp-c's performance drops; it is lower than Paxos because some requests take 3 RTTs. Skyros offers 2× higher throughput than Paxos and Curp-c.

5.7.2 Augmenting with Commutativity. While Skyros offers performance advantages over Curp-c in many cases, non-nilext updates can reduce the performance of Skyros. Curp-c can complete such operations in 1 RTT (when they do not conflict). Figure 14(e)-no-conflict case shows this: with 10% non-nilext writes, Curp-c performs better than Skyros.

Fortunately, however, nil-externality is compatible with commutativity. We build Skyros-Comm, a variant of Skyros that exploits commutativity to speed up non-nilext operations. Skyros-Comm handles nilext writes and reads in the same way as Skyros. However, non-nilext writes are handled similar to Curp-c. Upon a non-nilext write, a replica checks for conflicts with the pending nilext and non-nilext writes. If there are none, similar to curp-c, the replicas add this operation to their durability logs. Since non-nilext operations expose state, the leader also executes the operation and returns the result. Clients wait for supermajority responses including the execution result from the leader and acknowledgments from other replicas. Similar to Skyros, these responses must be from the same view.

Skyros-Comm handles non-nilext-write conflicts in 2 or 3 RTTs. A conflicting non-nilext write at the leader is treated similar to a read that accesses a pending update, finishing the operation in 2 RTTs. If the conflict does not arise at the leader but at the followers, the client detects the conflict and resends the request to the leader. The leader then enforces order by committing the request (and prior ones) to other replicas, finishing the operation in a total of 3 RTTs. Note that Skyros-Comm does not check for conflicts for nilext writes because they are ordered and executed only lazily.

The last bar in Figure 14(e)-no-conflict case shows that Skyros-Comm matches Curp-c's performance because it commits non-nilext writes faster than Skyros. Figure 14(e)-zipfian case shows that Curp-c's performance reduces due to conflicts. Skyros performs similar to Curp-c because of the 10% non-nilext writes. Skyros-Comm, however, improves performance over Skyros and Curp-c by combining the advantages of nil-externality and commutativity.

6 Discussion

In this paper, we exploit nilext interfaces in the context of leader-based replication for key-value stores. Further, our evaluation focused on single-datacenter settings. However, the general idea of exploiting nil-externality can be applied in other contexts as well. We discuss such possible extensions.

Beyond Key-value Stores. Key-value stores (especially ones built atop write-optimized structures) have many nilext interfaces, enabling fast replication. Nil-externality can be exploited to perform fast replication for other systems such as databases and file systems as well. As an example, consider the POSIX file API. Writes in POSIX (i.e., the write system call, and variants like pwrite and O_APPEND writes) are nilext because they do not externalize state, barring catastrophic I/O errors (e.g., due to a bad disk). Writes can thus be replicated performantly. Further, some file systems have been built upon write-optimized structures [26, 39], making most file-system operations nilext by design. A nilext-aware protocol can enable fast replication for such file systems.

Leaderless Protocols. Skyros is a leader-based protocol. The leader can become a performance bottleneck in such leader-based protocols. Also, clients cannot make progress when the leader fails (before a new leader is chosen). Leaderless protocols [54, 58] allow any replica to accept requests, leading to better performance and availability. The idea of exploiting nil-externality can be applied to such leaderless protocols as well. Leaderless protocols such as EPaxos [58] exploit commutativity to commit requests in one WAN RTT in geo-replicated settings. However, conflicting writes incur additional roundtrips. Such a protocol can be augmented to exploit nil-externality to avoid resolving conflicts on nilext writes and do so only on non-nilext writes or reads.

Multi Datacenter Settings. Unlike protocols designed for the data center [50, 67], Skyros is applicable to geo-replicated settings as well. By avoiding one WAN RTT, Skyros can reduce latency for nilext operations significantly. However, in

some scenarios, SKYROS may lead to higher latencies than a traditional 2-RTT protocol. In particular, when a majority of the replicas (but not a supermajority) are in the same region as the client, committing to a majority in two RTTs might be cheaper than committing to a supermajority in one RTT. While such a deployment is not commonly used (for fault tolerance reasons), when it is, SKYROS could be modified to fall back to the "slow" 2-RTT protocol based on measurements (similar to recent systems [78]).

7 Related Work

Commit Before Externalize. Our idea of deferring work until externalization bears similarity to prior systems. Xsyncfs defers disk I/O until output is externalized [60], essentially moving the output commit [25, 72] to clients. SpecPaxos [67], Zyzzyva [42], and SpecBFT [77] do the same for replication. As discussed in §3.4.1, these protocols execute requests in the correct order before notifying the end application. Our approach, in contrast, defers ordering or executing nilext operations beyond notifying the end application.

State modified by nilext updates can be externalized by later non-nilext operations upon which SKYROS enforces the required ordering and execution. Occult [56] and CAD [31] use a similar idea at a high-level. Occult defers enforcing causal consistency upon writes and does so only when clients read data. Similarly, CAD does not guarantee durability when writes complete; writes are made durable only upon subsequent reads [31]. However, these systems do not offer linearizability unlike SKYROS. Further, these systems defer work on all updates unlike our work which defers work based on whether or not the write is nilext. Prior work in unreplicated databases [30] realizes that some transactions only return an abort or commit and thus can be evaluated lazily, improving performance. Our work focuses on replicated storage and identifies a general interface-level property that allows deferring ordering and execution.

Exploiting Semantics. Inconsistent replication (IR) [80] realizes that *inconsistent* operations only require durability, and thus can be completed in 1 RTT. Nilext operations, in contrast, require durability and ordering. Further, IR cannot support general state machines. Prior replication [45, 58, 64] and transaction protocols [59] use commutativity to improve performance. Nil-externality has advantages over and combines well with commutativity (§5.7). SKYROS's use of DAG to resolve real-time order has a similar flavor to commutative protocols [58, 59]). However, these protocols resolve order in the common-case before execution; SKYROS needs such a step only during view changes. Gemini [49] and Pileus [43] realize that some operations need only weak consistency and perform these operations faster; we focus on realizing strong consistency with high performance.

SMR Optimizations. Apart from the approaches in §3.4.1, prior systems have pushed consensus into the network [20, 21]. Domino uses a predictive approach to reduce latency in WAN [78] and allows clients to choose between Multi-Paxos and Fast-Paxos schemes. As discussed in §6, ideas from Domino can be utilized in SKYROS to fall back to a 2-RTT path in geo-replicated scenarios where a single RTT to a supermajority is more expensive than two RTTs to a majority. Prior work has also proposed other techniques to realize high performance in multi-core servers [34, 40], by enabling quorum reads [12], and by partitioning state [47]. Such optimizations could also benefit SKYROS.

Local Storage Techniques. Techniques in SKYROS bear similarities to database write-ahead logging (WAL) [57] and file-system journaling [35]. However, our techniques differ in important aspects. While WAL and journaling do enable delaying writes to final on-disk pages, the writes are still applied to in-memory pages before responding to clients. Further, background disk writes are not triggered by externalizing operations but rather occur asynchronously; externalizing operations can proceed by accessing the in-memory state. In contrast, SKYROS defers applying updates altogether until externalization. While both WAL and the durability log in SKYROS ensure durability, WAL also imposes an order of transactions. Group commit [23, 35] batches several updates to amortize disk-access costs; Multi-Paxos and SKYROS similarly use batching at the leader to amortize cost.

8 Conclusion

In this paper, we identify nil-externality, a storage-interface property, and show that this property is prevalent in storage systems. We design nilext-aware replication, a new approach to replication that takes advantage of nilext interfaces to improve performance by lazily ordering and executing updates. We experimentally demonstrate that nilext-aware replication improves performance over existing approaches for a range of workloads. More broadly, our work shows that exposing and exploiting properties across layers of a storage system can bring significant performance benefit. Storage systems, today, layer existing replication protocols upon local storage systems (such as key-value stores). Such black-box layering masks vital information across these layers, resulting in missed performance opportunities. This paper shows that by making the replication layer aware of the underlying storage-interface properties, higher performance can be realized.

The source code of SKYROS and our experimental artifacts are available at https://bitbucket.org/aganesan4/skyros/.

Acknowledgments. We thank Bernard Wong (our shepherd) and the anonymous SOSP '21 reviewers for their insightful comments. We thank the following VMware Research Group members for their invaluable discussions: Jon Howell, Lalith Suresh, Marcos Aguilera, Mihai Budiu, Naama Ben-David, Rob Johnson, and Sujata Banerjee. Finally, the first two authors would like to extend special thanks to grandmother Jayanthy Alagappan for taking care of their toddler daughter while they were working on this paper.

References

[1] 2021. Memcached Commands. https://github.com/memcached/memcached/wiki/Commands#set.

[2] Hussam Abu-Libdeh, Robbert Van Renesse, and Ymir Vigfusson. 2013. Leveraging Sharding in the Design of Scalable Replication Protocols. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '13)*. Santa Clara, CA.

[3] Apache. 2021. ZooKeeper. https://zookeeper.apache.org/.

[4] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. 1995. Sharing Memory Robustly in Message-passing Systems. *Journal of the ACM (JACM)* 42, 1 (1995), 124–142.

[5] AWS News Blog. 2020. New EC2 M5zn Instances – Fastest Intel Xeon Scalable CPU in the Cloud. https://aws.amazon.com/blogs/aws/new-ec2-m5zn-instances-fastest-intel-xeon-scalable-cpu-in-the-cloud/.

[6] Michael A Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C Kuszmaul, Donald E Porter, Jun Yuan, and Yang Zhan. 2015. An Introduction to Be-trees and Write-optimization. *USENIX ;login:* 40, 5 (2015), 22–28.

[7] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. 2011. Paxos Replicated State Machines As the Basis of a High-performance Data Store. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI '11)*. Boston, MA.

[8] Gerth Stølting Brodal and Rolf Fagerberg. 2003. Lower Bounds for External Memory Dictionaries.. In *SODA*, Vol. 3.

[9] Navin Budhiraja, Keith Marzullo, Fred B Schneider, and Sam Toueg. 1993. The Primary-backup Approach. *Distributed systems* 2 (1993).

[10] Matthew Burke, Audrey Cheng, and Wyatt Lloyd. 2020. Gryff: Unifying Consensus and Shared Registers. In *Proceedings of the 17th Symposium on Networked Systems Design and Implementation (NSDI '20)*. Santa Clara, CA.

[11] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*. Santa Clara, CA.

[12] 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)*. Renton, WA.

[13] David R Cheriton. 1987. UIO: A Uniform I/O System Interface for Distributed Systems. *ACM Transactions on Computer Systems (TOCS)* 5, 1 (1987).

[14] Austin T Clements, M Frans Kaashoek, Nickolai Zeldovich, Robert T Morris, and Eddie Kohler. 2013. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*. Farmington, Pennsylvania.

[15] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. 2020. SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. Online.

[16] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '10)*. Indianapolis, IA.

[17] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally Distributed Database. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI '12)*. Hollywood, CA.

[18] James Cowling and Barbara Liskov. 2012. Granola: Low-overhead Distributed Transaction Coordination. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA.

[19] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. 2006. HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*. Seattle, WA.

[20] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. 2020. P4xos: Consensus as a Network Service. *IEEE/ACM Transactions on Networking* 28, 4 (2020).

[21] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. 2015. NetPaxos: Consensus at Network Speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR '15)*. Santa Clara, CA.

[22] Denis Serenyi. [n. d.]. Cluster-Level Storage @ Google. http://www.pdsw.org/pdsw-discs17/slides/PDSW-DISCS-Google-Keynote.pdf.

[23] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A. Wood. 1984. Implementation Techniques for Main Memory Database Systems. In *Proceedings of the 1984 ACM SIGMOD Conference on the Management of Data (SIGMOD '84)*. Boston, MA.

[24] Effi Ofer, Danny Harnik, and Ronen Kat. 2021. Object Storage Traces: A Treasure Trove of Information for Optimizing Cloud Workloads. https://www.ibm.com/cloud/blog/object-storage-traces.

[25] Elmootazbellah Nabil Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B Johnson. 2002. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)* 34, 3 (2002), 375–408.

[26] John Esmet, Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul. 2012. The TokuFS Streaming File System. In *4th Workshop on Hot Topics in Storage and File Systems (HotStorage '12)*. Boston, Massachussetts.

[27] Facebook. 2016. MyRocks: A space- and write-optimized MySQL database. https://engineering.fb.com/2016/08/31/core-data/myrocks-a-space-and-write-optimized-mysql-database/.

[28] Facebook. 2021. Merge Operator. https://github.com/facebook/rocksdb/wiki/Merge-Operator.

[29] Facebook. 2021. RocksDB. http://rocksdb.org/.

[30] Jose M Faleiro, Alexander Thomson, and Daniel J Abadi. 2014. Lazy Evaluation of Transactions in Database Systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. Snowbird, UT.

[31] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2020. Strong and Efficient Consistency with Consistency-aware Durability. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*. Santa Clara, CA.

[32] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*. Bolton Landing, New York.

[33] Sanjay Ghemawhat, Jeff Dean, Chris Mumford, David Grogan, and Victor Costan. 2011. LevelDB. https://github.com/google/leveldb.

[34] Zhenyu Guo, Chuntao Hong, Mao Yang, Dong Zhou, Lidong Zhou, and Li Zhuang. 2014. Rex: Replication at the Speed of Multi-core. In *Proceedings of the EuroSys Conference (EuroSys '14)*. Amsterdam, The Netherlands.

[35] Robert Hagmann. 1987. Reimplementing the Cedar File System using Logging and Group Commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP '87)*. Austin, Texas.

[36] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990).

[37] Paul Hudak. 1989. Conception, Evolution, and Application of Functional Programming Languages. *ACM Computing Survey* 21, 3 (1989).

[38] IBM. 2021. Locations for Resource Deployment: Multizone Regions. https://cloud.ibm.com/docs/overview?topic=overview-locations#mzr-table.

[39] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. 2015. BetrFS: A Right-optimized Write-optimized File System. In *Proceedings of the 14th USENIX Symposium on File and Storage Technologies (FAST '15)*. Santa Clara, CA.

[40] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. 2012. All About Eve: Execute-verify Replication for Multi-core Servers. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI '12)*. Hollywood, CA.

[41] Bettina Kemme, Fernando Pedone, Gustavo Alonso, and André Schiper. 1999. Processing transactions over optimistic atomic broadcast protocols. In *International Symposium on Distributed Computing (DISC 99)*. Bratislava, Slovak Republic.

[42] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2007. Zyzzyva: Speculative Byzantine Fault Tolerance. In *ACM SIGOPS Operating Systems Review*, Vol. 41. ACM, 45–58.

[43] Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Doug Terry. 2013. Consistency-based Service Level Agreements for Cloud Storage. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*. Farmington, Pennsylvania.

[44] Leslie Lamport. 2001. Paxos Made Simple. *ACM Sigact News* 32, 4 (2001), 18–25.

[45] Leslie Lamport. 2005. Generalized Consensus and Paxos. (2005).

[46] Butler W Lampson. 1983. Hints for Computer System Design. In *Proceedings of the 9th ACM Symposium on Operating System Principles (SOSP '83)*. Bretton Woods, New Hampshire.

[47] Long Hoang Le, Enrique Fynn, Mojtaba Eslahi-Kelorazi, Robert Soulé, and Fernando Pedone. 2019. Dynastar: Optimized Dynamic Partitioning for Scalable State Machine Replication. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS '19)*. Dallas, TX.

[48] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John Ousterhout. 2015. Implementing Linearizability at Large Scale and Low Latency. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*. Monterey, California.

[49] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI '12)*. Hollywood, CA.

[50] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. 2016. Just Say No to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*. Savannah, GA.

[51] Wei Lin, Mao Yang, Lintao Zhang, and Lidong Zhou. 2008. *PacificA: Replication in Log-based Distributed Storage Systems.* Technical Report MSR-TR-2008-25.

[52] Barbara Liskov and James Cowling. 2012. Viewstamped Replication Revisited. (2012).

[53] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. 1991. Replication in the Harp file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*. Pacific Grove, CA.

[54] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. 2008. Mencius: Building Efficient Replicated State Machines for WANs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI '08)*. San Diego, CA.

[55] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. MyRocks: LSM-tree Database Storage Engine Serving Facebook's Social Graph. *Proceedings of the VLDB Endowment* 13, 12 (2020).

[56] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. 2017. I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades. In *Proceedings of the 14th Symposium on Networked Systems Design and Implementation (NSDI '17)*. Boston, MA.

[57] Chandrasekaran Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks using Write-Ahead Logging. *ACM Transactions on Database Systems (TODS)* 17, 1 (1992), 94–162.

[58] Iulian Moraru, David G Andersen, and Michael Kaminsky. 2013. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*. Farmington, Pennsylvania.

[59] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. 2016. Consolidating Concurrency Control and Consensus for Commits under Conflicts. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*. Savannah, GA.

[60] Edmund B Nightingale, Kaushik Veeraraghavan, Peter M Chen, and Jason Flinn. 2006. Rethink the sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*. Seattle, WA.

[61] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. 2013. Scaling Memcache at Facebook. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI '13)*. Lombard, IL.

[62] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA.

[63] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (1996).

[64] Seo Jin Park and John Ousterhout. 2019. Exploiting Commutativity For Practical Fast Replication. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI '19)*. Boston, MA.

[65] Fernando Pedone and André Schiper. 2002. Handling Message Semantics with Generic Broadcast Protocols. *Distributed Computing* (2002).

[66] Percona. 2013. Fast Updates with TokuDB. https://www.percona.com/blog/2013/02/12/fast-updates-with-tokudb/.

[67] Dan RK Ports, Jialin Li, Vincent Liu, Naveen Kr Sharma, and Arvind Krishnamurthy. 2015. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI '15)*. Oakland, CA.

[68] Sudip Roy, Lucja Kot, and Christoph Koch. 2013. Quantum databases. In *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research (CIDR 2013)*. Asilomar, CA.

[69] Stephen M Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K Ousterhout. 2011. It's Time for Low Latency.. In *The Thirteenth Workshop on Hot Topics in Operating Systems (HotOS XIII)*. Napa, CA.

[70] Russel Sandberg. 1986. The Sun Network File System: Design, Implementation and Experience. In *Proceedings of the USENIX Summer Technical Conference (USENIX Summer '86)*. Atlanta, GA.

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

[72] Rob Strom and Shaula Yemini. 1985. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems (TOCS)* 3, 3 (1985), 204–226.

[73] Amy Tai, Andrew Kryczka, Shobhit O. Kanaujia, Kyle Jamieson, Michael J. Freedman, and Asaf Cidon. 2019. Who's Afraid of Uncorrectable Bit Errors? Online Recovery of Flash Errors with Distributed Redundancy. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA.

[74] Twitter. 2012. Caching with Twemcache. https://blog.twitter.com/engineering/en_us/a/2012/caching-with-twemcache.html.

[75] Twitter. 2020. Twitter Cache Trace. https://github.com/twitter/cache-trace.

[76] Robbert Van Renesse and Fred B Schneider. 2004. Chain Replication for Supporting High Throughput and Availability. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*. San Francisco, CA.

[77] Benjamin Wester, James A Cowling, Edmund B Nightingale, Peter M Chen, Jason Flinn, and Barbara Liskov. 2009. Tolerating Latency in Replicated State Machines Through Client Speculation.. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI '09)*. Boston, MA.

[78] Xinan Yan, Linguan Yang, and Bernard Wong. 2020. Domino: Using Network Measurements to Reduce State Machine Replication Latency in WANs. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*.

[79] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A Large Scale Analysis of Hundreds of In-memory Cache Clusters at Twitter. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI '20)*. Banff, Canada.

[80] Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan RK Ports. 2015. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*. Monterey, California.