Harmonia: Near-Linear Scalability for Replicated Storage with In-Network Conflict Detection

Hang Zhu
Johns Hopkins University
hzhu@jhu.edu

Zhihao Bai
Johns Hopkins University
zbai1@jhu.edu

Jialin Li
University of Washington
lijl@cs.washington.edu

Ellis Michael
University of Washington
emichael@cs.washington.edu

Dan R. K. Ports
Microsoft Research
dan@drkp.net

Ion Stoica
UC Berkeley
istoica@cs.berkeley.edu

Xin Jin
Johns Hopkins University
xinjin@cs.jhu.edu

ABSTRACT

Distributed storage employs replication to mask failures and improve availability. However, these systems typically exhibit a hard tradeoff between consistency and performance. Ensuring consistency introduces coordination overhead, and as a result the system throughput does not scale with the number of replicas. We present Harmonia, a replicated storage architecture that exploits the capability of new-generation programmable switches to obviate this tradeoff by providing near-linear scalability without sacrificing consistency. To achieve this goal, Harmonia detects read-write conflicts in the network, which enables any replica to serve reads for objects with no pending writes. Harmonia implements this functionality at line rate, thus imposing no performance overhead. We have implemented a prototype of Harmonia on a cluster of commodity servers connected by a Barefoot Tofino switch, and have integrated it with Redis. We demonstrate the generality of our approach by supporting a variety of replication protocols, including primary-backup, chain replication, Viewstamped Replication, and NOPaxos. Experimental results show that Harmonia improves the throughput of these protocols by up to 10× for a replication factor of 10, providing near-linear scalability up to the limit of our testbed.

PVLDB Reference Format:
Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan R. K. Ports, Ion Stoica, and Xin Jin. Harmonia: Near-Linear Scalability for Replicated Storage with In-Network Conflict Detection. *PVLDB*, 13(3): 375-388, 2019.
DOI: https://doi.org/10.14778/3368289.3368301

1. Introduction

Replication is one of the fundamental tools in the modern distributed storage developer's arsenal. Failures are a regular appearance in large-scale distributed systems, and strongly consistent replication can transparently mask these faults to achieve high system availability. However, it comes with a high performance cost.

Proceedings of the VLDB Endowment, Vol. 13, No. 3
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3368289.3368301

One might hope that adding more servers to a replicated system would increase not just its reliability but also its system performance—ideally, providing *linear scalability* with the number of replicas. The reality is quite the opposite: performance decreases with more replicas, as an expensive replication protocol needs to be run to ensure that all replicas are consistent. Despite much effort to reduce the cost of these protocols, the best case is a system that approaches the performance of a single node [70, 42].

Can we build a strongly consistent replicated system that approaches linear scalability? Write operations inherently need to be applied to all replicas, so more replicas cannot increase the write throughput. However, many real-world workloads are highly skewed towards reads [30, 53]—with read:write ratios as high as 30:1 [9]. A scalable but naive approach is to allow *any individual replica* to serve a read, permitting the system to achieve near-linear scalability for such workloads. Yet this runs afoul of a fundamental limitation. Individual replicas may lag behind, or run ahead of, the consensus state of the group. Thus, serving reads from any storage replica has the potential to return stale or even uncommitted data, compromising the consistency guarantee of the replicated system. Addressing this at the protocol level requires protocol-specific modifications that invariably involve extra coordination – e.g., an extra phase for each write in CRAQ [69] – leading to substantial performance overhead.

We show that it is possible to circumvent this limitation and simultaneously achieve near-linear scalability and consistency for replicated storage. We do so with Harmonia,[1] a new replicated storage architecture that exploits the capability of new-generation programmable switches. Our key observation is that while individual replicas may constantly diverge from the consensus state, the set of *inconsistent data* at any given time is small. A storage system may store millions or billions of objects or files. Of these, only the ones that have writes in progress—i.e., the set of objects actively being updated—may be inconsistent. For the others, any replica can safely serve a read. Two features of many storage systems make this an especially powerful observation: (i) read-intensive workloads in real-world deployments [53, 9] mean that fewer objects are written over time, and (ii) emerging in-memory storage systems [56, 5, 4, 71] complete writes faster, reducing the interval of inconsistency.

The challenge in leveraging this insight lies in efficiently detecting which objects are dirty, i.e., have pending updates. Implement-

[1]Harmonia draws its name from the ancient Greek goddess of concord, i.e., *lack of conflict*.

ing this functionality in a server would make the system be bottlenecked by the server, instead of scaling out with the number of storage replicas. Harmonia demonstrates that this idea can be realized on-path in the network with programmable switches at line rate, with no performance penalties. The core component of Harmonia is a read-write conflict detection module in the switch data plane that monitors all requests to the system and tracks the dirty set. The switch detects whether a read conflicts with pending writes, and if not, the switch sends it directly to one of the replicas. Other requests are executed according to the normal protocol. This design exploits two principal benefits of in-network processing: (i) the central location of the switch on the data path that allows it to monitor traffic to the cluster, and (ii) its capability for line-rate, near-zero overhead processing.

Harmonia is a general approach. It augments existing replication protocols with the ability to serve reads from any replica, and does not sacrifice fault tolerance or strong consistency (i.e., linearizability). As a result, it can be applied to both major classes of replication protocols—primary-backup and quorum-based. We have applied Harmonia to a range of representative protocols, including primary-backup [17], chain replication [70], Viewstamped Replication [54, 45], and NOPaxos [42].

In summary, this paper demonstrates that:

- The Harmonia architecture can achieve near-linear scalability with near-zero overhead by moving conflict detection to an in-network component. (§5, §6)
- The Harmonia conflict detection mechanism can be implemented in the data plane of new-generation programmable switches and run at line rate. (§7)
- Many replication protocols can be integrated with Harmonia while maintaining linearizability. (§8)

We implement a Harmonia prototype using a cluster of servers connected by a Barefoot Tofino switch and integrate it with Redis. Our experiments show that Harmonia improves the throughput of the replication protocols by up to 10× for a replication factor of 10, providing near-linear scalability up to the limit of our testbed. We provide a proof of correctness and a model-checked TLA+ specification in a technical report [74]. Of course, the highest possible write throughput is that of *one* replica, since writes have to be processed by all replicas. This can be achieved by chain replication [70] and NOPaxos [42]. Harmonia fills in the missing piece for reads: it demonstrates how to make reads scalable without sacrificing either write performance or consistency.

2. The Case for Programmable Switches in Database Clusters

Cluster database systems are built using clusters of commodity servers connected via an Ethernet network. They rely on distributed protocols to locate data and ensure its availability—notably, replication protocols like chain replication [70] or Paxos [37] to tolerate failures. As the performance of single-node databases systems increases—a result of faster storage technologies and domain-specific accelerators—the relative cost of coordination becomes an increasingly substantial part of the workload.

Traditionally, these systems treat the network merely as a "dumb" transport that conveys packets without regard for their meaning. A recent hardware trend allows rethinking this assumption. *Programmable switches* expose the high-speed processing power of switch ASICs to new, application-specific protocols. These devices, now commercially available from vendors like Barefoot, Cavium, Mellanox, and Broadcom, can achieve similar performance and cost to their non-programmable counterparts. These offer the capability to (1) parse user-defined headers; (2) reconfigure the packet processing pipeline with custom logic; and (3) maintain state between packets; while still maintaining full line-rate (e.g., 64×100Gbit) throughput.

Harmonia uses a programmable switch to track the state of a replicated database and route requests differently based on whether conflicting operations are present. It can be viewed as a new take on network anycast [57], a network primitive that routes packets to the closest suitable host; Harmonia extends this to operate based on application semantics, namely read/write conflicts, rather than simple proximity. As we demonstrate, the resource requirements imposed on the network device are modest. In this respect, it differs from recent proposals to move storage [36, 35, 26, 25] or query operators [40, 50, 65] themselves to programmable switches. These applications require repurposing nearly all the computational and storage resources on a switch (potentially interfering with existing networking functionality [62]), and yet still face major restrictions caused by resource limits (e.g., limiting storage to 128-byte objects [36, 35]). Harmonia's lighter-weight approach allows it either to complement such systems or to be deployed in more general cluster scenarios.

3. The Quest for Scalable Replication

Harmonia improves the scalability of replication protocols that provide *linearizability* [32]. These make it appear as though operations are being executed, one at a time, on a single replica. As we will see, existing protocols impose a performance cost for providing this guarantee.

3.1 Replication Protocols

Many replication protocols can be used to ensure this property. They fall primarily into two classes—primary-backup protocols and quorum-based protocols.

Primary-backup protocols. The primary-backup protocol [17] organizes a system into a *primary* replica, which is responsible for determining the order and results of operations, and a set of *backup* replicas that execute operations as directed by the primary. This is typically achieved by having the primary transfer state updates to the replicas after operation execution. At any time, only one primary is in operation. Should it fail, one of the backup replicas is promoted to be the new primary—a task often mediated by an external configuration service [34, 18] or manual intervention. The primary-backup protocol is able to tolerate f node failures with f+1 nodes.

The primary-backup protocol has many variants. Chain replication [70] is a high-throughput variant used in many storage systems [6, 59, 29]. It organizes the replicas into a chain. Writes are sent to the head and propagated to the tail; reads are directly processed by the tail. The system throughput is bounded by a single server—the tail.

Quorum-based protocols. Quorum-based protocols such as Paxos [37, 38] operate by ensuring that each operation is executed at a quorum—typically a majority—of replicas before it is considered successful. While they seem quite different from primary-backup protocols, the conceptual gap is not as wide as it appears in practice. Many Paxos deployments use the Multi-Paxos optimization [38] (or, equivalently, Viewstamped Replication [54] and Raft [55]). One of the replicas runs the first phase of the protocol to elect itself as a stable *leader* until it fails. It can then run the second phase repeatedly to execute multiple operations and commit to other replicas. System throughput is largely determined by the number of messages that need to be processed by the bottleneck

node, i.e., the leader. A common optimization allows the leader to execute reads without coordinating with the others, by giving the leader a lease. Ultimately, however, the system throughput is limited to that of one server.

3.2 Towards Linear Scalability

The replication protocols above can achieve, at best, the throughput of a single server. They allow reads to be processed by one designated replica—the tail in chain replication or the leader in Multi-Paxos. That single replica then becomes the bottleneck. Read scalability, i.e., making system throughput scale with the number of replicas, requires going further.

Could we achieve read scalability by allowing reads to be processed by *any* replica, not just a single designated one, without coordination? Unfortunately, naively doing so could violate consistency. Updates cannot be applied instantly across all the replicas, so at any given moment some of the replicas may not be consistent. We categorize the resulting anomalies into two kinds.

Read-ahead anomalies. A read-ahead anomaly occurs when a replica returns a result that has not yet been committed. This might reflect a future state of the system, or show updates that will never complete. Neither is correct. Reading uncommitted data is possible if reads are sent to any replica and a replica answers a read with its latest known state, which may contain data that is not committed. We use chain replication as an example. Specifically, suppose there are three nodes, and the latest update to an object has been propagated to nodes 1 and 2. A read on this object sent to either of these nodes would return the new value, but a request to node 3 would still return the old value—which could cause a client to see an update appearing and disappearing depending on which replica it contacts. Simultaneously ensuring read scalability and linearizability thus requires ensuring that clients only read committed data, regardless of which replica they contact.

Read-behind anomalies. One might hope that these anomalies could be avoided by requiring replicas to return the latest *known committed* value. Unfortunately, this introduces a second class of anomalies, where some replicas may return a stale result that does not reflect the latest updates. This too is a violation of linearizability. Consider a Multi-Paxos deployment, in which replicas only execute an operation once they have been notified by the leader that consensus has been reached for that operation. Suppose that a client submits a write to an object, and consider the instant just after the leader receives the last response in a quorum. It can then execute the operation and respond to the client. However, the other replicas do not know that the operation is successful. If the client then executes a read to one of the other replicas, and it responds—unilaterally—with its latest committed value, the client will not see the effect of its latest write.

Protocols. We classify replication protocols based on the anomalies. We refer to protocols that have each type of anomalies as *read-ahead protocols* and *read-behind protocols*, respectively. In this paper, primary-backup and chain replication are read-ahead protocols, and Viewstamped Replication/Multi-Paxos and NOPaxos are read-behind protocols. Note that although the primary-backup systems are read-ahead and the quorum systems are read-behind, this is not necessarily true in general; read-ahead quorum protocols are also possible, for example.

4. Harmonia Approach

How, then, can we *safely* and *efficiently* achieve read scalability, without sacrificing linearizability? The key is to view the system at the *individual object level*. At any given time, the majority of objects are quiescent, i.e., have no modifications in progress. These objects will be consistent across all the replicas. In that case, any replica can unilaterally answer a read for the object. While modifications to an object *are* in progress, reads on the object must follow the full protocol.

Conceptually, Harmonia achieves read scalability by introducing a new component to the system, a request scheduler. The request scheduler monitors requests to the system to detect conflicts between read and write operations. Abstractly, it maintains a table of objects in the system and tracks whether they are contended or uncontended, i.e., the *dirty set*. When there is no conflict, it directs reads to any replica. The request is flagged so that the replica can respond directly. When conflicts are detected, i.e., a concurrent write is in progress, reads follow the normal protocol.

To allow the request scheduler to detect conflicts, it needs to be able to interpose on all read and write traffic to the system. This means that the request scheduler must be able to achieve very high throughput—implementing the conflict detection in a server would still make the entire system be bottlenecked by the server. Instead, we implement the request scheduler in the network itself, leveraging the capability of programmable switches to run at line rate, imposing no performance penalties.

Conflict detection has been used before to achieve read scalability for certain replicated systems. Specifically, CRAQ [69] provides read scalability for chain replication by tracking contended and uncontended objects at the protocol level. This requires introducing an extra phase to notify replicas when an object is clean vs. dirty. Harmonia's in-switch conflict detection provides two main benefits. First, it generalizes the approach to support different protocols—as examples, we have used Harmonia with primary-backup, chain replication, Viewstamped Replication, and NOPaxos. Supporting the diverse range of protocols in use today is important because they occupy different points in the design space: latency-optimized vs. throughput-optimized, read-optimized vs. write-optimized, storage overhead vs. performance under failure, etc. CRAQ is specific to chain replication, and it is not clear that it is possible to extend its protocol-level approach to other protocols. Second, Harmonia's in-switch implementation avoids additional overhead of tracking the dirty set. As we show in Section 10.5, CRAQ is able to achieve read scalability only at the expense of a decrease in write throughput. Harmonia has no such cost.

4.1 Challenges

Translating the basic model of the request scheduler above to a working system implementation presents several technical challenges as follows.

1. How can we expose system state to the request scheduler so that it can implement conflict detection?
2. How do we ensure the switch's view of which objects are contended matches the system's reality, even as messages are dropped or reordered by the network? Errors here may cause clients to see invalid results.
3. How do we implement this functionality fully within a switch data plane with limited computational and storage capacity?
4. What modifications are needed to replication protocols to ensure they provide linearizability when integrated with Harmonia?

5. Harmonia Architecture

Harmonia is a new replicated storage architecture that achieves near-linear scalability without sacrificing consistency using in-network conflict detection. This is implemented using an in-switch request scheduler, which is located on the path between the clients

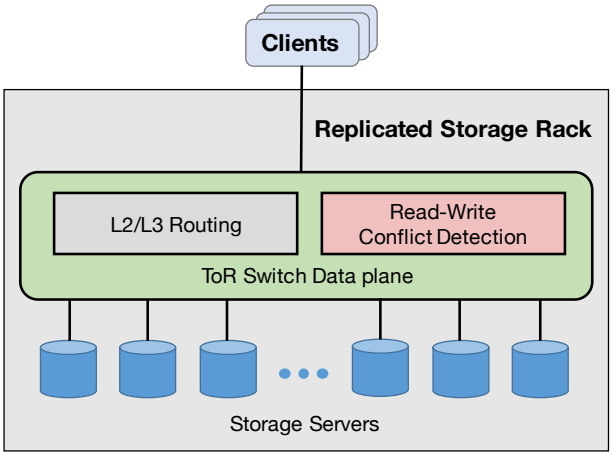


Figure 1: Harmonia architecture.

and server nodes. In many enterprise and cloud scenarios where storage servers are located in a dedicated storage rack, this can be conveniently achieved by using the top-of-rack (ToR) switch, as shown in Figure 1. We discuss other options in §7.3.

Switch. The switch implements the Harmonia request scheduler. It is responsible for detecting read-write conflicts. It behaves as a standard L2/L3 switch, but provides additional conflict detection functionality for packets with a reserved L4 port. This makes Harmonia fully compatible with normal network functionality.

The *read-write conflict detection module* identifies whether a read has a conflict with a pending write. It does this by maintaining a sequence number, a dirty set and the last-committed point (§6). We show how to handle requests with this module while *guaranteeing consistency* (§6), and how to use the register arrays to design a hash table supporting the necessary operations *at line rate* (§7). Given the switch on-chip memory is limited, the key insight is that the dirty set of writes is small and only the object IDs, instead of the object content, need to be stored.

While the Harmonia switch can be rebooted or replaced and is not a single point of failure of the storage system, there is only a single active Harmonia switch for conflict detection at any time. The replication protocol enforces this invariant by periodically agreeing on which switch to use for each time slice (§6.3).

Storage servers. The storage servers store objects and serve requests, using a replication protocol for consistency and fault tolerance. Harmonia requires minimal modifications to the replication protocol (§8). It incorporates a shim layer in each server to translate Harmonia request packets to API calls to the storage system.

Clients. Harmonia provides a client library for applications to access the storage system, which provides a similar interface as existing storage systems. e.g., `GET` and `SET` for Redis [5]. The library translates between API calls and Harmonia packets. It exposes two important fields in the packet header to Harmonia switch: the operation type (read or write), and the affected object ID.

6. In-Network Conflict Detection

Key idea. Harmonia employs a switch as a conflict detector, which tracks the dirty set, i.e., the set of contended objects. While the available switch memory is limited, the set of objects with outstanding writes is small compared to the overall storage size of the system, making this set readily implementable on the switch.

Algorithm 1 ProcessRequestSwitch(pkt)

- seq: sequence number at switch
- $dirty\_set$: map containing largest sequence number for each object with pending writes
- $last\_committed$: largest known committed sequence number

```
1:  if pkt.op == WRITE then
2:      seq ← seq + 1
3:      pkt.seq ← seq
4:      dirty_set.put(pkt.obj_id, seq)
5:  else if pkt.op == WRITE-COMPLETION then
6:      if pkt.seq ≥ dirty_set.get(pkt.obj_id) then
7:          dirty_set.delete(obj_id)
8:      last_committed ← max(last_committed, pkt.seq)
9:  else if pkt.op == READ then
10:     if ¬dirty_set.contains(pkt.obj_id) then
11:         pkt.last_committed ← last_committed
12:         pkt.dst ← random replica
13: Update packet header and forward
```

To implement conflict detection, a Harmonia switch tracks three pieces of state: (i) a *monotonically-increasing sequence number*,[2] which is incremented and inserted into each write, (ii) a *dirty set*, which additionally tracks the largest sequence number of the outstanding writes to each object, and (iii) the *last-committed point*, which tracks the sequence number of the latest write committed by the system known to the switch.

The dirty set allows the switch to detect when a read contends with ongoing writes. When they do not, Harmonia can send the read to a single random replica for better performance. Otherwise, these reads are passed unmodified to the underlying replication protocol. The sequence number disambiguates concurrent writes to the same object, permitting the switch to store *only one* entry per contended object in the dirty set. The last-committed sequence number is used to ensure linearizability in the face of reordered messages, as will be described in §6.2.

We now describe in detail the interface and how it is used. Implementing it in the switch data plane is described in §7. We use the primary-backup protocol as an example in this section, and describe adapting other protocols in §8.

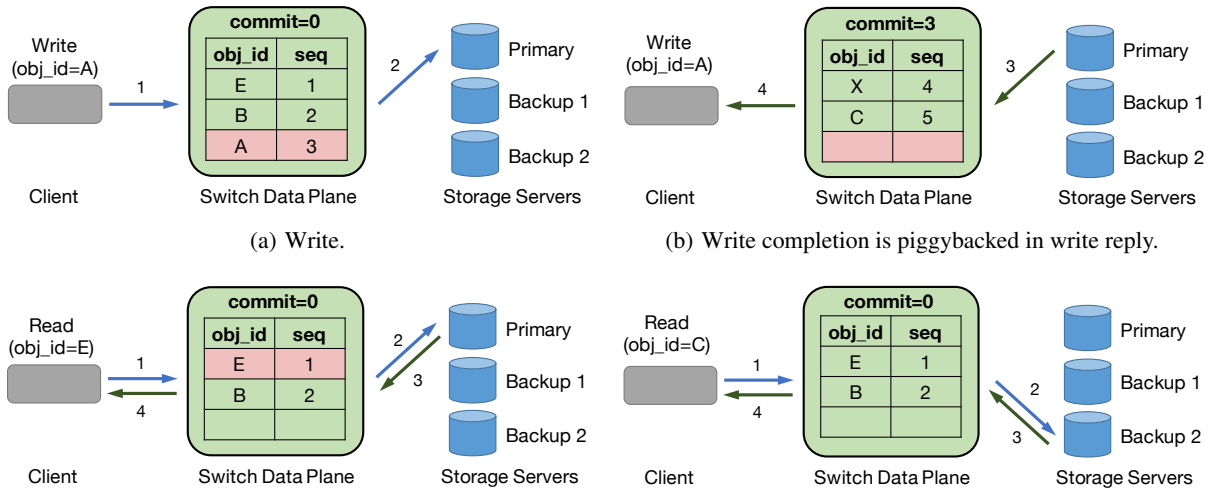
6.1 Basic Request Processing

The Harmonia in-switch request scheduler processes three types of operations: READ, WRITE, and WRITE-COMPLETION. For each replicated system, the switch is initialized with the addresses of the replicas and tracks the three pieces of state described above: the dirty set, the monotonically-increasing sequence number, and the sequence number of the latest committed write. The handling of a single request is outlined in pseudo code in Algorithm 1.

Writes. All writes are assigned a sequence number by Harmonia. The objects being written and the assigned sequence numbers are added to the dirty set in the switch (lines 1–4).

Write completions. Write completions are sent by the replication protocol once a write is fully committed. If a write is the last outstanding write to the object, the object is removed from the dirty set in the switch. The last-committed sequence number maintained by the switch is then updated (lines 5–8).

[2]We use the term sequence number here for simplicity. Sequentiality is not necessary; a strictly increasing timestamp would suffice.



(a) Write.

(b) Write completion is piggybacked in write reply.

(c) Read and reply on object with pending writes.

(d) Read and reply on object without pending writes.

Figure 2: Handling different types of storage requests.

Reads. Reads are routed by the switch, either through the normal replication protocol or to a randomly selected replica, based on whether the object being read is contended or not. The switch checks whether the object ID is in the dirty set. If so, the switch sends the request unmodified, causing it to be processed according to the normal replication protocol; otherwise, the read is sent to a random replica for better performance (lines 9–12). The request is also stamped with the last-committed sequence number on the switch for linearizability, as will be discussed in §6.2.

Example. Figure 2 shows an example workflow. Figure 2(a) and 2(b) show a write operation. The switch adds `obj_ID=A` to the dirty set when it observes the write. It removes the object ID upon the write completion, which can be piggybacked in the write reply, and updates the last-committed sequence number. What is in the dirty set determines how reads are handled. In Figure 2(c), the read is for object `E`, which has pending writes, so the request is sent to the primary for guaranteeing consistency. On the other hand, in Figure 2(d), object `C` is not in the dirty set, so it is sent to the second backup for better performance.

6.2 Handling Network Asynchrony

In an ideal network, where messages are processed in order, only using the dirty set would be sufficient to guarantee consistency. In a real, asynchronous network, just because a read to an object was uncontended when the request passed through the switch does not mean it will still be so when the request arrives at a replica: the message might have been delayed so long that a new write to the same object has been partially processed. Harmonia avoids this using the sequence number and last-committed point.

Write order requirement. The key invariant of the dirty set requires that an object not be removed until *all* concurrent writes to that object have been completed. Since the Harmonia switch only keeps track of the largest sequence number for each object, Harmonia requires that the replication protocol processes writes only in sequence number order. This is straightforward to implement in a replication protocol, e.g., via tracking the last received sequence number and discarding any out-of-order writes.

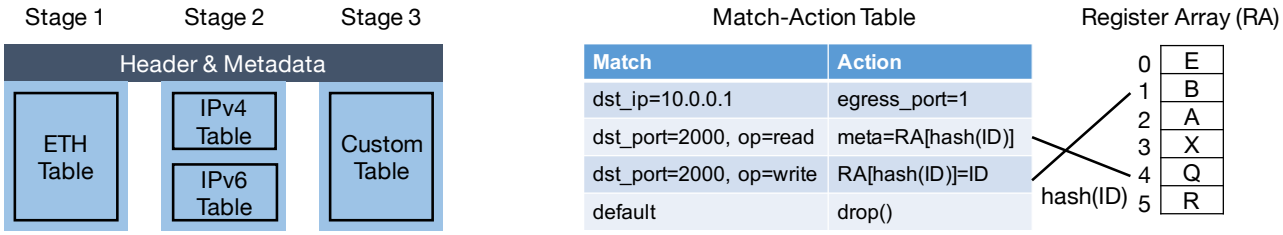
Dropped messages. If a WRITE-COMPLETION or forwarded WRITE message is dropped, an object may remain in the dirty set indefinitely. While in principle harmless—it is always safe to consider an uncontended object dirty—it may cause a performance degradation. However, because writes are processed in order, any stray entries in the dirty set can be removed as soon as a WRITE-COMPLETION message with a higher sequence number arrives. These stray objects can be removed by the switch as it processes reads (i.e., by removing the object ID if its sequence number in the dirty set is less than or equal to the last committed sequence number). This removal can also be done periodically.

Last-committed point for linearizability. Harmonia can *maintain linearizability*, even when the network arbitrarily delays or reorders packets. The switch uses its dirty set to ensure that a single-replica read does not contend with ongoing writes *at the time it is processed by the switch*. This is not sufficient to entirely eliminate inconsistent reads. However, the last-committed sequence number stamped into the read will provide enough information for the recipient to compute whether or not processing the read locally is safe. In the primary-backup, a write after a read on the same object would have a *higher* sequence number than the last-committed point carried in the read. As such, a backup can detect the conflict even if the write happens to be executed at the backup before the read arrives, and then send the read to the primary for linearizability. Detailed discussion on adapting protocols is presented in §8.

6.3 Failure Handling

Failure handling is critical to replicated systems. Harmonia would be of limited utility if the switch were a single point of failure. However, because the switch only keeps *soft state* (i.e., the dirty set, the sequence number and the last-committed point), it can be rebooted or replaced. The Harmonia failure handling protocol restores the ability for the new switch to send requests through the normal case first, and then restores the single-replica read capability, limiting the system downtime to a minimum. As such, the switch is not a single point of failure, and can be safely replaced without sacrificing consistency.

Handling switch failures. When the switch fails, the operator either reboots it or replaces it with a backup switch. While the switch only maintains soft state, care must be taken to preserve consistency during the transition. As in prior systems [42, 41], the sequence numbers in Harmonia are augmented with the switch's unique ID, which is monotonically increasing when assigned to switches, and ordered lexicographically considering the switch's ID first. This ensures that no two writes have the same sequence number. Next, be-



(a) Switch multi-stage packet processing pipeline. (b) Example custom table.

Figure 3: Switch data plane structure.

fore a newly initialized switch can process writes, Harmonia must guarantee that single-replica reads issued by the previous switch will not be processed. Otherwise, in read-behind protocols, the previous switch and a lagging replica could bilaterally process a read without seeing the results of the latest writes, resulting in read-behind anomalies. To prevent these anomalies, the replication protocol periodically agrees to allow single-replica reads from the current switch for a time period. Before allowing the new switch to send writes, the replication must agree to refuse single-replica reads with smaller switch IDs, and either the previous switch's time should expire or all replicas should agree to cut it short. This technique is similar in spirit to the leases used as an optimization to allow leader-only reads in many protocols. Finally, once the switch receives its first WRITE-COMPLETION with the new switch ID, both its last-committed point and dirty set will be up to date, and it can safely send single-replica reads.

In a multi-rack deployment (§7.3) when the switch is not a ToR switch, traffic can be directly rerouted to another switch without waiting for rebooting or replacing the failed switch. In this scenario, the switch that handles the rerouted traffic becomes the Harmonia switch for the corresponding replica group. It uses a bigger switch ID and the aforementioned process to ensure consistency.

Handling server failures. The storage system handles a server failure based on the replication protocol, and notifies the switch control plane at the beginning and end of the process. The switch first removes the failed replica from the replica addresses in the data plane, so that following requests would not be scheduled to it. After the failed replica is recovered or a replacement server is added, the switch adds the corresponding address to the replica addresses, enabling requests to be scheduled to the server.

7. Data Plane Design and Implementation

Can the Harmonia approach be supported by a real switch? We answer this in the affirmative by showing how to implement it for a programmable switch [15, 14] (e.g., Barefoot's Tofino [11]), and evaluate its resource usage.

7.1 Data Plane Design

The in-network conflict detection module is implemented in the data plane of a modern programmable switch. The sequence number and last-committed point can be stored in two registers, and the dirty set can be stored in a hash table implemented with register arrays. While previous work has shown how to use the register arrays to store key-value data in a switch [36, 35], our work has two major differences: (i) the hash table only needs to store object IDs, instead of both IDs and values; (ii) the hash table needs to support insertion, search and deletion operations at line rate, instead of only search. We provide some background on programmable switches, and then describe the hash table design.

Switch data plane structure. Figure 3 illustrates the basic data plane structure of a modern programmable switching ASIC. The packet processing pipeline contains multiple stages, as shown in Figure 3(a). Packets are processed by the stages one after one. Match-action tables are the basic element used to process packets. If two tables have no dependencies, they can be placed in the same stage, e.g., IPv4 and IPv6 tables in Figure 3(a).

A match-action table contains a list of rules that specifies how packets are processed, as shown in Figure 3(b). A match in a rule specifies a header pattern, and the action specifies how the matched packets should be processed. For example, the first rule in Figure 3(b) forwards packets to egress port 1 for packets with destination IP 10.0.0.1. Each stage also contains register arrays that can be accessed at line rate. Programmable switches allow developers to define custom packet formats and match-action tables to realize their own protocols. The example in Figure 3(b) assumes two custom fields in the packet header, which are `op` for operation and `ID` for object ID. The second and third rules perform read and write on the register array based on `op` type, and the index of the register array is computed by the hash of `ID`.

Developers use a domain-specific language such as P4 [14] to write a program for a custom data plane, and then use a complier to compile the program to a binary that can be loaded to the switch. Each stage has resource constraints on the size of match-action tables (depending on the complexity of matches and actions) and register arrays (depending on the length and width).

Multi-stage hash table with register arrays. The switch data plane provides basic constructs for the conflict detection module. A register array can be naturally used to store the object IDs. We can use the hash of an object ID as the index of the register array, and store the object ID as the value in the register slot. One challenge is to handle hash collisions, as the switch can only perform a limited, fixed number of operations per stage. Collision resolution for hash tables is a well-studied problem, and the multi-stage structure of the switch data plane makes it natural to implement open addressing techniques to handle collisions. Specifically, we allocate a register array in each stage and use different hash functions for different stages. In this way, if several objects collide in one stage, they are less likely to collide in another stage. Figure 4 shows the design.

- **Insertion.** For a write, the object ID is inserted to the first stage with an empty slot for the object (Figure 4(a)). The write is dropped if no slot is available.
- **Search.** For a read, the switch iterates over all stages to see if any slot contains the same object ID (Figure 4(b)).
- **Deletion.** For a write completion, the switch iterates over all stages and removes the object ID (Figure 4(c)).

Variable-length object IDs. Many systems use *variable-length* IDs, e.g., variable-length keys in key-value stores and file paths in file systems. Due to switch limitations, Harmonia must use *fixed-length* object IDs for conflict detection. However, variable-length

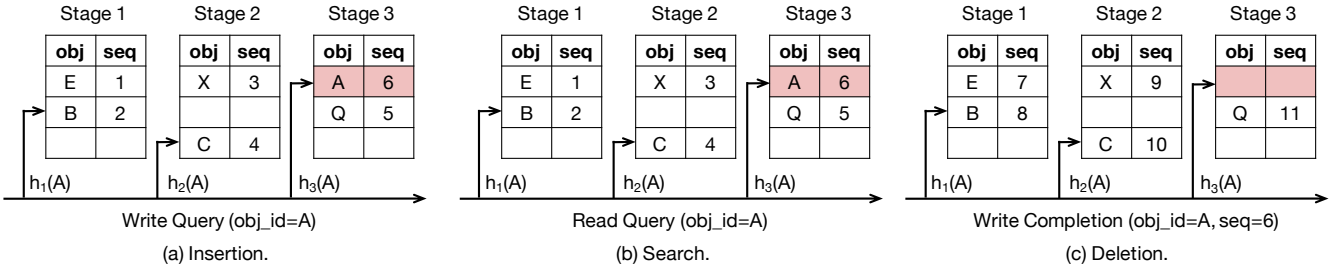


Figure 4: Multi-stage hash table design that supports insertion, search and deletion in the switch data plane.

IDs can be accommodated by having the clients store fixed-length hashes of the original ID in the Harmonia packet header; the original ID is sent in the packet payload. Harmonia then uses the fixed-length hashes for conflict detection. Hash collisions may degrade performance but cannot introduce consistency issues; they can only cause Harmonia to believe a key is contended, not vice versa.

Protocol support. Harmonia is at application level, and thus is compatible with existing L2-L4 network protocols. The network forwards Harmonia packets as other packets using L2/L3 protocols. Harmonia embeds a Harmonia header inside the L4 (UDP/TCP) payload. The header only carries two fields (i.e., the operation type and the object ID) from clients, and is additionally inserted the sequence number field for writes by the Harmonia switch. Only the Harmonia switch needs to implement conflict detection and request scheduling, which is invoked by a reserved L4 port for Harmonia packets; other switches do not need to understand or implement Harmonia, and simply treat Harmonia packets as normal packets.

UDP is widely used by many key-value stores for low latency [56, 5, 4, 71]. Harmonia supports UDP by embedding the Harmonia header inside the UDP payload. A read request only requires one packet, and a write request may span several packets in order to carry large values for update. To support multi-packet writes, we use two operation types, namely, `WFirst` and `WRest`, to distinguish between the first and rest packets of a write. Packets with `WFirst` are processed as single-packet writes, i.e., causing the object IDs to be added to the dirty set of the switch; those with `WRest` are directly forwarded. This avoids the object ID to be added to the dirty set by multiple times for the same write.

Some key-value stores use TCP for *reliable writes* [56, 5, 4, 71]. This can be supported by Harmonia in a similar way as multi-packet UDP writes. The switch forwards the TCP packets (including SYN, FIN, and actual request) based on the normal replication protocol, e.g., to the primary for primary-backup. In the actual request packet, the client embeds the Harmonia header in the TCP payload, so that the Harmonia switch can add the object ID to the dirty set. Similar to multi-packet UDP writes, `WFirst` and `WRest` are used for large values that require multiple packets to transmit. Supporting TCP reads is not straightforward because an uncontended read can be forwarded to any replica, which needs to be consistent for all packets of the same read, and whether the read is contended or not cannot be decided when the switch sees the TCP SYN packet. This can be addressed by requiring the TCP SYN packet to also include the Harmonia header in the TCP payload, and using a deterministic hash function based on the packet header to pick replicas for uncontended reads.

7.2 Resource Usage

Switch on-chip memory is a limited resource. Will there be enough memory to store the entire dirty set of pending writes? Our key insight is that since the switch only performs conflict detection, it does not need to store actual data, but only the object IDs. This is in contrast to previous designs like NetCache [36] and NetChain [35] that use switch memory for object storage directly. Moreover, while the storage system can store a massive number of objects, the number of writes at any given time is small, implying that the dirty set is far smaller than the storage size.

Suppose we use n stages and each stage has a register array with m slots. Let the hash table utilization be u to account for hash collisions. The switch can support up to unm writes at a given time. Suppose the duration of each write is t, and the write ratio is w. Then the switch can support unm/t writes per second—or a total throughput of $unm/(wt)$—before exhausting memory. As a concrete example, let n=3, m=64000, u=50%, t=1 ms and w=5%. The switch can support a write throughput of 96 million requests per second (MRPS), and a total throughput of 1.92 billion requests per second (BRPS). Let both the object ID and sequence number be 32 bits. It only consumes 1.5MB memory. Given that a commodity switch has 10–20 stages and a few tens of MB memory [67, 36, 35], this example only conservatively uses a small fraction of switch memory.

In worst cases (e.g., many in-flight writes), the limited switch memory can be full. The switch drops writes when there are no empty slots, and the clients can queue and retry the dropped requests after a timeout. It is critical for the switch to drop these writes to guarantee linearizability. Otherwise, these writes would be sent to servers without being tracked, and reads on the corresponding objects would be sent to any replica, violating linearizability. While dropping writes could limit the throughput, the above back-of-the-envelope calculation shows the switch only needs 1.5MB memory to support 1.92 BRPS, and the evaluation (Figure 13(a)) empirically shows that the switch has sufficient memory to track in-flight writes and the memory is not the system bottleneck.

7.3 Deployment Issues

We imagine two possible deployment scenarios for Harmonia. First, it can be easily integrated with clustered storage systems, as shown in Figure 1. All servers are deployed in the same rack, allowing the ToR switch to be the central location that sees all the storage traffic. This is practical to many real-world use cases, including on-premise storage clusters for enterprises and specialized storage clusters in the cloud. It is easy to deploy as it only needs to add Harmonia's functionality to the ToR switch of the storage rack, and does not need to change other switches.

For cloud-scale storage, replicas may be distributed among many different racks for fault tolerance. Placing the Harmonia scheduler on a ToR switch, which only sees storage traffic to its own rack, does not suffice. Instead, we leverage a network serialization approach [42, 61], where all traffic destined for a replica group is redirected through a designated switch. This solution incurs minimal drawbacks on performance. First, as shown in prior work [42], with careful selection of the switch (e.g., a spine switch in a two-layer leaf-spine network), this need not increase latency. Second,

because different replica groups can use different switches as their request schedulers and the capacity of a switch far exceeds that of a single replica group, this does not reduce throughput. Importantly, the designated switch is not a single point of failure, as a switch failure only affects the replica groups it is responsible for, and the traffic can be directly rerouted to another switch while guaranteeing consistency as described in §6.3.

8. Adapting Replication Protocols

Safely using a replication protocol with Harmonia imposes three responsibilities on the protocol. It must:

1. process writes only in sequence number order;
2. allow single-replica reads only from one active switch at a time;
3. ensure that single-replica reads for uncontended objects still return linearizable results.

Responsibility (1) can be handled trivially by dropping messages that arrive out of order, and responsibility (2) can be implemented in the same manner as leader leases in traditional replication protocols. We therefore focus on responsibility (3) here. How this is handled is different for the two categories of read-ahead and read-behind protocols.

To demonstrate its generality, we apply Harmonia to representative protocols from both classes. We explain the necessary protocol modifications and give a brief argument for correctness. A full proof of correctness and a model-checked TLA+ specification are in a technical report [74].

8.1 Requirements for Linearizability

Let us first specify the requirements that must be satisfied for a Harmonia-adapted protocol to be correct. All write operations are processed by the replication protocol based on the sequence number order. We need only, then, consider the read operations. The following two properties are sufficient for linearizability.

- **P1. Visibility.** A read operation sees the effects of all write operations that finished before it started.
- **P2. Integrity.** A read operation will not see the effects of any uncommitted write operation at the time the read finished.

In the context of Harmonia, read operations follow the normal-case replication protocol if they refer to an object in the dirty set, and hence we need only consider the fast-path read operations. For these, P1 can equivalently be stated as follows.

- **P1. Visibility**. The replication protocol must only send a completion notification for a write to the scheduler if any subsequent single-replica read sent to any replica will reflect the effect of the write operation.

8.2 Read-Ahead Protocols

Both primary-backup and chain replication are read-ahead protocols that cannot have read-behind anomalies, because they only reply to the client once an operation has been executed on all replicas. As a result, they inherently satisfy P1. We adapt them to send a WRITE-COMPLETION notification to the switch at the same time as responding to the client.

However, read-ahead anomalies *are* possible: reads naively executed at a single replica can reflect uncommitted results. We use the last-committed sequence number provided by the Harmonia switch to prevent this. When a replica receives a fast-path read for object o, it checks that the last-committed sequence number attached to the request is at least as large as the latest write applied to o. If not, it forwards the request to the primary or tail, to be executed using the normal protocol. Otherwise, this implies that all writes to o processed by the replica were committed at the time the read was handled by the switch, satisfying P2.

8.3 Read-Behind Protocols

We have applied Harmonia to two quorum protocols: Viewstamped Replication [54, 45], a leader-based consensus protocol equivalent to Multi-Paxos [38] or Raft [55], and NOPaxos [42], a network-aware, single-phase consensus protocol. Both are read-behind protocols. Because replicas in these protocols only execute operations once they have been committed, P2 is trivially satisfied.

Furthermore, because the last committed point in the Harmonia switch is greater than or equal to the sequence numbers of all writes removed from its dirty set, replicas can ensure visibility (P1) by rejecting (and sending to the leader for processing through the normal protocol) all fast-path reads whose last committed points are larger than that of the last locally *committed and executed* write. In read-behind protocols, WRITE-COMPLETIONs can be sent along with the response to the client. However, in order to reduce the number of rejected fast-path reads, we delay WRITE-COMPLETIONs until the write has likely been executed on all replicas.

Viewstamped replication. For Viewstamped Replication, we add an additional phase to operation processing that ensures a quorum of replicas have *committed and executed* the operation. Concurrently with responding to the client, the VR leader sends a COMMIT message to the other replicas. Our additional phase calls for the replicas to respond with a COMMIT-ACK message.[3] Only once the leader receives a quorum of COMMIT-ACK messages for an operation with sequence number n does it send a $\langle$WRITE-COMPLETION, *object_id*, $n\rangle$ notification.

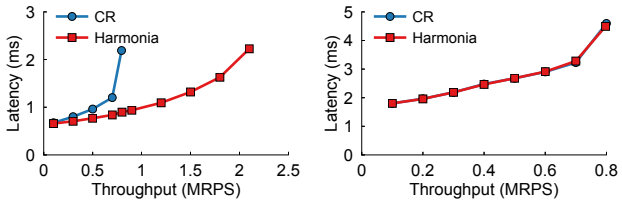
NOPaxos. NOPaxos [42] uses an in-network sequencer to enable a single-round, coordination-free consensus protocol. It is a natural fit for Harmonia, as both the sequencer and Harmonia's request scheduler can be deployed in the same switch. Although the replicas do not coordinate while handling operations, they already run a periodic synchronization protocol to ensure that they have executed a common, consistent prefix of the log [43] that serves the same purpose as the additional phase in VR. The only Harmonia modification needed is for the leader, upon completion of a synchronization, to send $\langle$WRITE-COMPLETION, *object_id*, *commit*$\rangle$ messages for all affected objects.

9. Implementation

We have implemented a Harmonia prototype and integrated it with Redis [5]. The switch data plane is implemented in P4 [14] and is compiled to Barefoot Tofino ASIC [11] with Barefoot P4 Studio [10]. There are multiple programmable switching chips, such as Barefoot Tofino [11], Broadcom Trident 3 & 4 [1], Cavium XPliant [20] and NetFPGA SUME [75]. We choose Barefoot Tofino for the prototype because it is publicly available and compatible with P4, provides comparable performance as Broadcom Trident 3 & 4, and is faster than NPU-based Cavium XPliant and FPGA-based NetFPGA SUME. We use 32-bit object IDs, and use 3 stages for the hash table. Each stage provides 64K slots to store the object IDs, resulting in a total of 192K slots for the hash table.

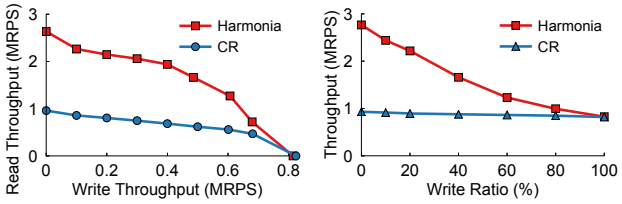
The shim layer in the storage servers is implemented in C++. It communicates with clients using Harmonia packets, and uses hiredis [2], which is the official C library of Redis [5], to read from and write to Redis. In additional to translate between Harmonia packets and Redis operations, the shim layers in the servers also communicate with each other to implement replication protocols. We have integrated Harmonia with multiple representative replication protocols (§10.5). We use the pipeline feature of Redis to batch

[3]These messages can be piggybacked on the next PREPARE and PREPARE-OK messages, eliminating overhead.



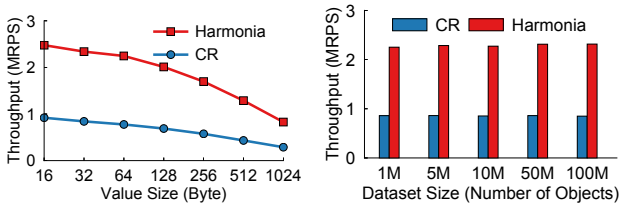
(a) Read-only workload. (b) Write-only workload.

Figure 5: Performance for reads and writes.



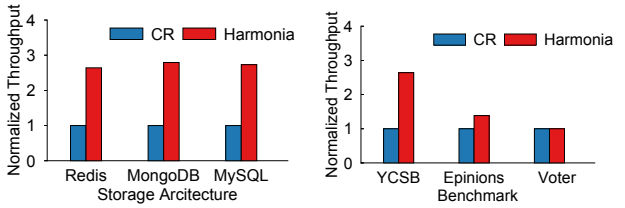
(a) Read vs. write. (b) Write ratio.

Figure 6: Performance for mixed workloads.



(a) Value size. (b) Dataset size.

Figure 7: Impact of value size and dataset size.



(a) Storage architecture. (b) Benchmark.

Figure 8: Performance under different storage architectures and benchmarks.

requests to Redis. Because Redis is single-threaded, we run eight Redis processes on each server to maximize per-server throughput. Our prototype achieves about 0.92 MQPS for reads and 0.8 MQPS for writes on a single server. The client library is implemented in C with Intel DPDK [3]. It generates requests to the storage system, and measures the system throughput and latency.

10. Evaluation

We show experimentally that Harmonia provides significant throughput improvements for read-mostly workloads, in combination with a variety of replication protocols, storage backends, workload parameters, and deployment scenarios.

Testbed. Our experiments are conducted on a testbed consisting of twelve server machines connected by a 6.5 Tbps Barefoot Tofino switch. Each server is equipped with an 8-core CPU (Intel Xeon E5-2620 @ 2.1GHz), 64 GB total memory, and one 40G NIC (Intel XL710). The server OS is Ubuntu 16.04.3 LTS. Ten storage servers run Redis v4.0.6 [5] as the storage backend; two generate client load using a DPDK-based workload generator.

Comparison. Redis is a widely-used open-source in-memory storage system. However, Redis does not provide native support for replication, only a cluster mode with weak consistency. We use a shim layer to implement several representative replication protocols, including primary-backup (PB) [17], chain replication (CR) [70], CRAQ [69] (a version of chain replication that makes reads more scalable at the cost of more expensive writes), Viewstamped Replication (VR) [54] and NOPaxos [42]. As described in §9, we run eight Redis processes on each server to maximize per-server throughput. The shim layer batches requests to Redis; the baseline (unreplicated) performance for one server is 0.92 MQPS for reads and 0.8 MQPS for writes.

We compare system performance with and without Harmonia for each protocol. Due to space constraints, we show the results of CR, a high-throughput variant of PB, in most figures; §10.5 compares performance across all protocols, demonstrating generality.

Workload. By default, we use three replicas and a workload of 5% write ratio based on YCSB [23] with uniform distribution on one million objects with 32-bit IDs and 128-bit values. The 5% write ratio is similar to that in real-world storage systems [53, 9], previous studies [47], and standard benchmarks like YCSB [23]. Harmonia supports atomic read-modify-write operations. The switch adds the object IDs to the dirty set when it receives the requests and removes the IDs when it receives the replies to ensure linearizability. In the experiments, we vary the parameters (write ratio, value size, dataset size, number of in-fight writes, storage system latency, number of replicas, and switch memory size), the storage architecture (Redis [5], MongoDB [48] and MySQL [51]), and the benchmark (YCSB [23], Epinions [28] and Voter [68]).

10.1 Latency vs. Throughput

We first conduct a basic throughput and latency experiment. The client generates requests to three replicas, and measures the average latency at different throughput levels.

Figure 5(a) shows the relationship between throughput and latency under a read-only workload. Since CR only uses the tail node to handle read requests, the throughput is bounded by that of one server. In comparison, since Harmonia uses the switch to detect read-write conflicts, it is able to fully utilize the capacity of all the three replicas when there are no conflicts. The read latency is a few hundred microseconds at low load, and increases as throughput goes up. For write-only workloads (Figure 5(b)), CR and Harmonia have identical performance, as Harmonia simply passes writes to the normal protocol.

To evaluate mixed workloads, the client fixes its rate of generating write requests, and measures the maximum read throughput that can be handled by the replicas. Figure 6(a) shows the read throughput as a function of write rate. Since CR can only leverage the capacity of the tail node, its read throughput is no more than that of one storage server, even when the write throughput is small. On the other hand, Harmonia can utilize all three replicas to handle reads when the write throughput is small. At low write rate, Harmonia improves the throughput by 3× over CR. At high write rate, both systems have similar throughput as Harmonia and CR process write requests in the same way. Figure 6(b) evaluates the performance for mixed workloads from another angle. The client fixes the ratio of writes and measures the saturated throughput. The figure shows the total throughput as a function of write ratio. Again, the throughput of CR is bounded by the tail node, while Harmonia can leverage all replicas to process reads. Similar to Figure 6(a), when the write ratio is high, Harmonia has little benefit as they process writes in the same way.

Figure 7 evaluates the impact of value size and dataset size. The throughput of both Harmonia and CR decreases with larger value

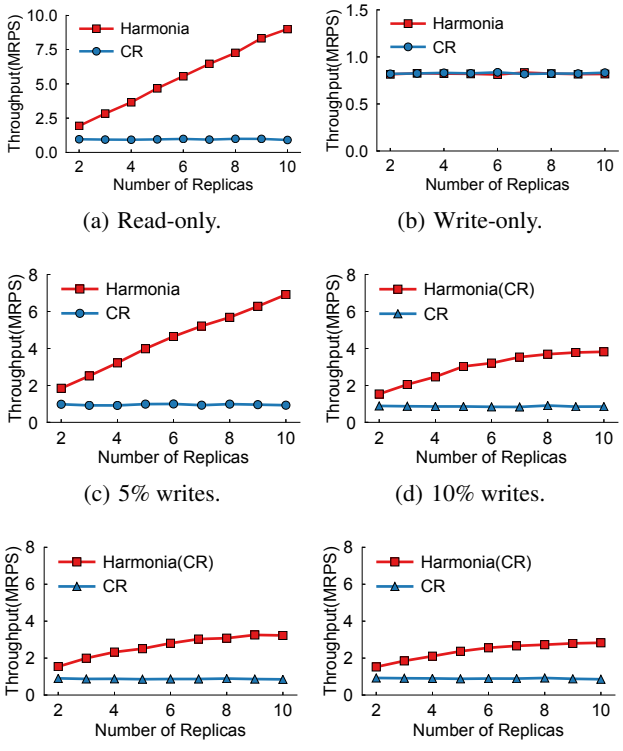


Figure 9: Total throughput with increasing numbers of replicas for three workloads.

size, as the servers spend more time on processing larger values. Because Harmonia only stores *object IDs*, instead of *object values*, in the switch, the value size does not affect Harmonia's ability to detect conflicts, and Harmonia outperforms CR regardless of the value size. Harmonia also outperforms CR under different dataset sizes, and the throughput of both systems is not affected by the dataset size in the evaluated range from 1 million to 100 million objects, because the servers in the testbed have sufficient memory to store all objects.

Besides Redis, our prototype also supports MongoDB [48] and MySQL [51] as storage engines. Figure 8(a) shows their throughput. The throughput of Harmonia is normalized to that of CR in each storage architecture to depict the improvement. The result demonstrates that Harmonia is a general approach that can be applied to different storage architectures. Figure 8(b) shows the normalized throughput under different benchmarks. Harmonia's performance depends on the read/write ratio of the workload: it provides a substantial performance improvement on the read-mostly (95% read) YCSB workload [23] and a modest one on the mixed (50%) Epinions [28] benchmark. Voter [68], a 100% write benchmark, is an example of a workload that Harmonia will not benefit, though neither does it impose a performance penalty.

10.2 Scalability

Harmonia offers near-linear read scalability for read-intensive workloads. We demonstrate this by varying the number of replicas and measuring system throughput in several representative cases. The scale is limited by the size of our twelve-server testbed: we can use up to ten servers as replicas, and two servers as clients to generate requests. Our high-performance client implementation written in C and DPDK is able to saturate ten replicas with two clients.

Harmonia offers dramatic improvements on read-only workloads (Figure 9(a)). For CR, increasing the number of replicas does not change the overall throughput, because it only uses the tail to handle reads. In contrast, Harmonia is able to utilize the other replicas to serve reads, causing throughput to increase linearly with the number of replicas. Harmonia improves the throughput by 10× with a replication factor of 10, limited by the testbed size. It can scale out until the switch is saturated. Multiple switches can be used for multiple replica groups to further scale out (§7.3). On write-only workloads (Figure 9(b)), Harmonia has no benefit regardless of the number of replicas because Harmonia uses the underlying replication protocol for writes. For CR, the throughput stays the same as more replicas are added since CR uses a chain to propagate writes.

Figure 9(c) considers throughput scalability under a mixed read-write workload with a write ratio of 5%. Again, CR does not scale with the number of replicas. In comparison, the throughput of Harmonia increases nearly linearly with the number of replicas. Under a read-intensive workload, Harmonia can efficiently utilize the remaining capacity on the other nodes. The total throughput here is smaller than that for read-only requests (Figure 9(a)), because handling writes is more expensive than handling reads and the tail node becomes the bottleneck as the number of replicas goes up to 10. Figure 9(d), Figure 9(e) and Figure 9(f) show the throughput scalability under 10%, 15% and 20% writes, respectively. Similarly, the throughput of Harmonia keeps increasing with more replicas, while that of CR does not scale. The throughput improvement is smaller with higher write ratio, because Harmonia uses in-network conflict detection to achieve read scalability, and does not change how the system handles writes.

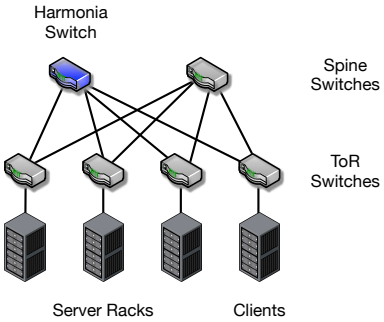


Figure 10: Topology for multi-rack experiments.

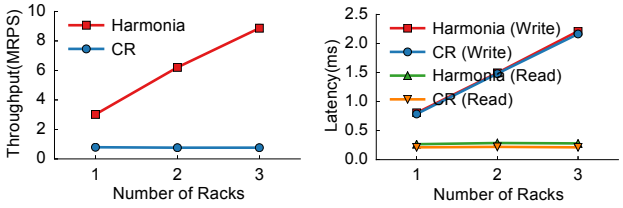
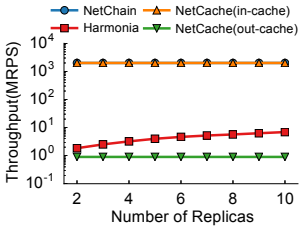


Figure 11: Performance with multiple racks.

In addition to single-rack deployments, we also consider the multi-rack deployment shown in Figure 10. This uses a different testbed consisting of four racks, each containing four servers (Xeon Silver 4114 at 2.2 GHz); up to three racks are used for Redis servers and the fourth for clients. Each rack has a single top-of-rack switch, and the racks are interconnected via 100 GbE links to two aggregation switches. We use one of the aggregation switches as the Harmonia switch, using a Tofino-based Arista 7170-64C; the other switches are Arista 7060CX-32 models, based on the (non-programmable) Broadcom Tomahawk chipset.

Figure 11(a) shows that Harmonia continues to provide scalability benefits as the number of racks increase. The tradeoff is that all traffic from clients to replicas must pass through the single Har-



(a) Performance.

	Value Size	Dataset Size
NetChain	≤128 Byte	≤ 64K
NetCache	≤128 Byte	Unlimited*
Harmonia	Unlimited*	Unlimited*

* Unlimited means the value size and data size are restricted by the storage system, not the switch.

(b) Value and dataset size.

Figure 12: Comparison with NetCache & NetChain.

monia switch. Though this does not pose a throughput bottleneck, there is a slight increase in latency for read and write operations (Figure 11(b)) caused by the routing constraint and the additional latency of the programmable switch. It remains, however, 1-2 order of magnitudes lower than the operation processing cost.

10.3 Comparison with In-Switch Storage

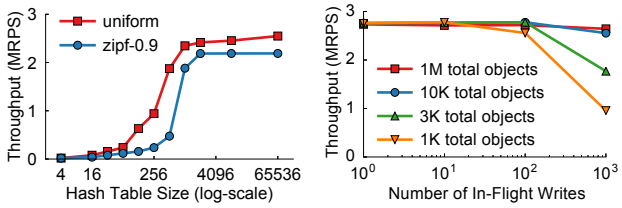
Two recent systems, NetCache [36] and NetChain [35], also use programmable switches, but in a different way: they store application data directly in the switch. These systems are able to achieve dramatically higher performance – as high as 4 *billion* operations per second (Figure 12(a)) – but subject to severe workload restrictions. As a result, they are designed for specific scenarios rather than the generic replicated storage that Harmonia targets: NetCache caches a small number of objects to alleviate hotspots and improve load balancing of a larger storage system, and NetChain provides replicated storage for a limited set of small objects, targeting configuration services and similar workloads.

Specifically, Figure 12(b) summarizes the supported workloads for these systems as compared to Harmonia. Both NetCache and NetChain are limited to values less than 128 bytes; most database workloads do not meet this constraint. On our workload, NetChain is forced to fall back to the much slower switch control processor, as it is limited to 64K objects (though for smaller workloads, it can achieve up to 2 billion QPS). NetCache integrates with a storage system that can handle larger datasets, but the cache itself can only hold a similar amount of data. These limitations are fundamental: on-die memory capacity and parse depth are major limiting factors in switch ASIC performance [15]. Harmonia avoids these limitations by storing only metadata in the switch.

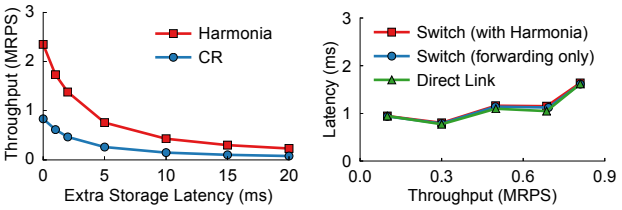
10.4 Switch Resource Usage and Overhead

Harmonia uses switch memory sparingly, as it only tracks metadata (object IDs and sequence numbers). In addition to the analytical evaluation (§7.2), we provide an empirical one by varying the size of the Harmonia switch's hash table, and measuring the total throughput of three replicas. Here, we use a write ratio of 5% and both uniform and skewed (zipf-0.9) request distributions across one million keys. As shown in Figure 13(a), Harmonia only requires about 2000 hash table slots to track all outstanding writes. Below this point, skewed workloads are more heavily affected by memory constraints: a hot object would always occupy a slot in the hash table, making the switch drop writes to other objects that collide on this slot, thus limiting throughput.

With 32-bit object IDs and 32-bit sequence numbers, 2000 slots only consume 16 KB memory. Given that commodity switches have tens of MB on-chip memory [67, 36, 35], the memory used by Harmonia only accounts for a tiny fraction of the total memory, e.g., only 1.6% (0.8%) for 10 MB (20 MB) memory. This result roughly matches the back-of-envelope calculations in §7.2, with differences coming from table utilization, write duration and total throughput. Thus, Harmonia can be added to the switch and

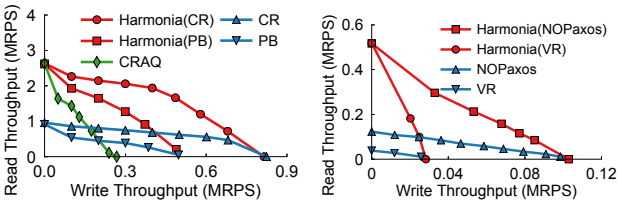


(a) Switch memory size. (b) In-flight writes.



(c) Storage latency. (d) Switch overhead.

Figure 13: Switch resource usage and overhead.



(a) Primary-backup. (b) Quorum-based.

Figure 14: Read throughput as write rate increases, for a variety of replication protocols.

co-exist with other modules without significant resource consumption. It also allows Harmonia to scale out to multiple replica groups with one switch, as one group only consumes little memory. This is especially important if Harmonia is deployed in a spine switch to support many replica groups across different racks.

Figure 13(b) shows the impact of in-flight writes. Here, we manually insert object IDs into the switch dirty set, representing in-flight writes and measure read throughput using a uniform access distribution. The throughput decreases with more in-flight writes, as contended reads can only be sent to the tail. This effect is smaller with larger dataset size, as the contended objects become a smaller subset of the total objects. One way this scenario might arise is if the backing store is slower (e.g., disk-based), causing writes to remain in flight longer. Figure 13(c) evaluates this effect more directly by adding extra latency to query processing to Redis. While the throughput decreases with longer storage latency, Harmonia still provides significant throughput improvements.

Figure 13(d) evaluates the switch packet processing overhead. It compares the latency and throughput of a Redis client and server, connected by either a direct link, a conventional switch, and a Harmonia switch. There is no significant difference, as the switch only incurs microsecond-level delay.

10.5 Generality

We show that Harmonia is a general approach by applying it to a variety of replication protocols. For each protocol, we examine throughput for a three-replica storage system with and without Harmonia. Figure 14 shows the read throughput as a function of write rate for different protocols. Figure 14(a) shows the results for two primary-backup protocols, PB and CR. Both PB and CR are limited by the performance of one server. Harmonia makes use of all three

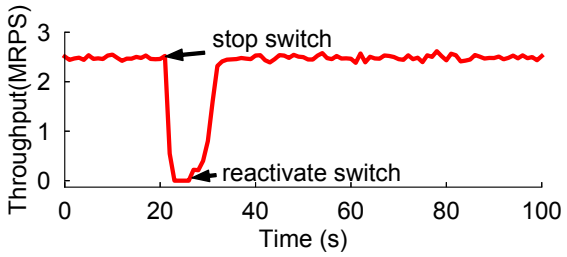


Figure 15: Failure handling result.

replicas to handle reads, and provides significantly higher throughput than PB and CR. CR is able to achieve higher write throughput than PB, as it uses a chain structure to propagate writes.

CRAQ, a modified version of CR, obtains higher read throughput than CR, as shown in Figure 14(a). This is because CRAQ allows reads to be sent to any replica (reads to dirty objects are forwarded to the tail). However, CRAQ adds an additional phase to write operations (first marking objects as dirty then committing the write). As a result, CRAQ's write throughput is much lower—hence the steeper curve. Harmonia (CR), which applies in-network conflict detection to CR, performs much better than CRAQ, achieving the same level of read scalability without degrading the performance of writes. Figure 14(b) shows the results for quorum-based protocols VR and NOPaxos. For faithful comparison, we use the original implementation of NOPaxos, including the middlebox-based sequencer prototype, which runs on a Cavium Octeon II network processor. We integrate Harmonia with these, rather than the Tofino switch and Redis-based backend. As a result, the absolute numbers in Figures 14(a) and 14(b) are incomparable. The trends, however, are the same. Harmonia significantly improves throughput for VR and NOPaxos.

10.6 Performance Under Failures

Finally, we show how Harmonia handles failures. To simulate a failure, we first manually stop and then reactivate the switch. Harmonia uses the mechanism described in §6.3 to correctly recover from the failure. Figure 15 shows the throughput during this period of failure and recovery. At time 20 s, we let the switch stop forwarding any packets, and the throughput drops to zero. We wait for a few seconds and then reactivate the switch to forward packets. Upon reactivation, the switch retains none of its former state and uses a new switch ID. The servers are notified with the new switch ID and agree to drop single-replica reads from the old switch. In the beginning, the switch forwards reads to the tail node and writes to the head node. During this time, the system throughput is the same as without Harmonia. After the first WRITE-COMPLETION with the new switch ID passes the switch, the switch has the up-to-date dirty set and last-committed point. At this time, the switch starts scheduling single-replica reads to the servers, and the system throughput is fully restored. Because the servers complete requests quickly, the transition time is minimal, and we can see that the system throughput returns to pre-failure levels within a few seconds.

11. Related Work

Replication protocols. Replication protocols are widely used by storage systems to achieve strong consistency and fault tolerance [33, 52, 66, 46, 58, 31, 7, 30, 22, 27, 8, 72, 12, 49, 19]. The primary-backup protocol [17] and its variations like chain replication [70] and CRAQ [69] assign replicas with different roles, and require operations to be executed by the replicas in a certain order. Quorum-based protocols [37, 63, 55, 54, 13] only require an operation to be executed at a quorum, instead of all replicas. While they do not distinguish the roles of replicas, they often employ an optimization that first elects a leader and then uses the leader to commit operations to other nodes. Vertical Paxos [39] proposes to incorporate these two classes of protocols into a single framework, by dividing a replication protocol into a steady state protocol and a reconfiguration protocol. CRAQ [69] is most similar in spirit to our work. It adapts chain replication to allow any replica to answer reads for uncontended objects by adding a second phase to the write protocol: objects are first marked dirty, then updated. Harmonia achieves the same goal without the write overhead by in-network conflict detection, and supports more general replication protocols.

Query scheduling. A related approach is taken in a line of database replication systems that achieve consistent transaction processing atop multiple databases, such as C-JDBC [21], FAS [64], and Ganymed [60]. These systems use a query scheduler to orchestrate queries among replicas with different states. The necessary logic is more complex for database transactions (and sometimes necessitates weaker isolation levels). Other systems, e.g., Breitbart *et al.* [16], use specific tree topologies to propagate updates lazily while maintaining serializablity, a conceptually similar approach to chain replication. Harmonia provides a near-zero-overhead scheduler implementation for replication using the network. Query processing also needs to be scheduled, and prior work has extended query planners to take into account network-level state. Xiong *et al.* [73] adapt a query planner to take into account available bandwidth and use traffic prioritization and bandwidth reservation to differentiate users. NetStore [24] selects the least congested path for transactions and caches data away from congested links.

In-network computing. The emerging programmable switches introduce new opportunities to move computation into the network. NetCache [36] and IncBricks [47] use in-network caching to improve the load balancing of key-value stores. NetChain [35] builds an in-network key-value store for coordination services. In each of these cases, because data is stored in switches themselves, both object size and dataset size are limited by the switch memory (Figure 12). SwitchKV [44] leverages programmable switches to realize content-based routing for load balancing in key-value stores. Eris [41] exploits programmable switches to realize concurrency control for distributed transactions. NetPaxos [26, 25] implements Paxos on switches. SpecPaxos [61] and NOPaxos [42] use switches to order messages to improve replication protocols. With NetPaxos, SpecPaxos and NOPaxos, reads still need to be executed by a quorum, or by a leader if the leader-based optimization is used. Harmonia improves these solutions by allowing reads not in the dirty set to be executed by any replica.

12. Conclusion

Harmonia is a new replicated storage architecture that achieves near-linear scalability and guarantees linearizability with in-network conflict detection. Harmonia leverages new-generation programmable switches to efficiently track the dirty set and detect read-write conflicts in the network data plane. Such a powerful capability enables Harmonia to safely schedule reads to the replicas without sacrificing consistency. Harmonia demonstrates that rethinking the division of labor between the network and end hosts makes it possible to achieve performance properties beyond the grasp of distributed systems alone.

Acknowledgments We thank the anonymous reviewers for their valuable feedback. This work is supported in part by NSF grants CRII-1755646, CNS-1813487 and CCF-1918757, Facebook Communications & Networking Research Award, and Amazon AWS Cloud Credits for Research Program.

13. REFERENCES

[1] Broadcom ethernet switches and switch fabric devices. https://www.broadcom.com/products/ethernet-connectivity/switching.

[2] Hiredis: Redis library. https://redis.io/.

[3] Intel data plane development kit (dpdk). http://dpdk.org/.

[4] Memcached key-value store. https://memcached.org/.

[5] Redis data structure store. https://redis.io/.

[6] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *ACM SOSP*, pages 1–14, 2009.

[7] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. *ACM Transactions on Computer Systems*, 29(5):109–126, 1996.

[8] Apache Hadoop Distributed File System (HDFS). http://hadoop.apache.org/.

[9] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS*, pages 53–64, 2012.

[10] Barefoot P4 Studio. https://www.barefootnetworks.com/products/brief-p4-studio/.

[11] Barefoot Tofino. https://www.barefootnetworks.com/technology/#tofino.

[12] D. Beaver, S. Kumar, H. C. Li, J. Sobel, P. Vajgel, et al. Finding a needle in Haystack: Facebook's photo storage. In *USENIX OSDI*, pages 1–8, 2010.

[13] K. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. *SIGOPS Operating Systems Review*, 21(5):123–138, 1987.

[14] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM CCR*, 44(3):87–95, 2014.

[15] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM*, pages 99–110, 2013.

[16] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update propagation protocols for replicated databases. In *ACM SIGMOD*, pages 97–108, 1999.

[17] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. In *Distributed systems*, volume 2, pages 199–216, 1993.

[18] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *USENIX OSDI*, pages 335–350, 2006.

[19] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure storage: A highly available cloud storage service with strong consistency. In *ACM SOSP*, pages 143–157, 2011.

[20] Cavium XPliant. https://www.cavium.com/.

[21] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-jdbc: Flexible database clustering middleware. In *USENIX ATC*, pages 9–18, 2004.

[22] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *USENIX OSDI*, pages 205–218, 2006.

[23] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *ACM Symposium on Cloud Computing*, pages 143–154, 2010.

[24] X. Cui, M. Mior, B. Wong, K. Daudjee, and S. Rizvi. NetStore: Leveraging network optimizations to improve distributed transaction processing performance. In *Proceedings of the Second International Workshop on Active Middleware on Modern Hardware*, pages 1–10, 2017.

[25] H. T. Dang, M. Canini, F. Pedone, and R. Soulé. Paxos made switch-y. *SIGCOMM CCR*, 46(2):18–24, 2016.

[26] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé. NetPaxos: Consensus at network speed. In *ACM SOSR*, page 5, 2015.

[27] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *ACM SOSP*, pages 205–220, 2007.

[28] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. OLTP-Bench: An extensible testbed for benchmarking relational databases. *PVLDB*, 7(4):277–288, 2013.

[29] R. Escriva, B. Wong, and E. G. Sirer. HyperDex: A distributed, searchable key-value store. In *ACM SIGCOMM*, pages 25–36, 2012.

[30] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *ACM SOSP*, pages 29–43, 2003.

[31] J. H. Hartman and J. K. Ousterhout. The Zebra striped network file system. *ACM Transactions on Computer Systems*, 13(3):274–310, 1995.

[32] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[33] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, 1988.

[34] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *USENIX ATC*, pages 145–158, 2010.

[35] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. NetChain: Scale-free sub-RTT coordination. In *USENIX NSDI*, pages 35–49, 2018.

[36] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing key-value stores with fast in-network caching. In *ACM SOSP*, pages 121–136, 2017.

[37] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[38] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[39] L. Lamport, D. Malkhi, and L. Zhou. Vertical paxos and primary-backup replication. In *ACM PODC*, pages 312–313, 2009.

[40] A. Lerner, R. Hussein, and P. Cudré-Mauroux. The case for network accelerated query processing. In *CIDR*, 2019.

[41] J. Li, E. Michael, and D. R. K. Ports. Eris: Coordination-free consistent transactions using in-network concurrency

control. In *ACM SOSP*, pages 104–120, 2017.

[42] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. Ports. Just say NO to Paxos overhead: Replacing consensus with network ordering. In *USENIX OSDI*, pages 467–483, 2016.

[43] J. Li, E. Michael, A. Szekeres, N. K. Sharma, and D. R. K. Ports. Just say NO to Paxos overhead: Replacing consensus with network ordering (extended version). Technical Report UW-CSE-TR-16-09-02, University of Washington CSE, Seattle, WA, USA, 2016.

[44] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman. Be fast, cheap and in control with SwitchKV. In *USENIX NSDI*, pages 31–44, 2016.

[45] B. Liskov and J. Cowling. Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, USA, July 2012.

[46] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp file system. In *ACM SOSP*, pages 226–238, 1991.

[47] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya. IncBricks: Toward in-network computation with an in-network cache. In *ACM ASPLOS*, pages 795–809, 2017.

[48] MongoDB. `https://www.mongodb.com/`.

[49] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, et al. f4: Facebook's warm BLOB storage system. In *USENIX OSDI*, pages 383–398, 2014.

[50] C. Mustard, F. Ruffy, A. Gakhokidze, I. Beschastnikh, and A. Fedorova. Jumpgate: In-network processing as a service for data analytics. In *USENIX HotCloud Workshop*, 2019.

[51] MySQL. `https://www.mysql.com/`.

[52] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, 1988.

[53] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *USENIX NSDI*, pages 385–398, 2013.

[54] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *ACM PODC*, pages 8–17, 1988.

[55] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC*, pages 305–319, 2014.

[56] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The RAMCloud storage system. *ACM Transactions on Computer Systems*, 33(3):7, 2015.

[57] C. Partridge, T. Mendez, and W. Milliken. Host anycasting service. RFC 1546, November 1993.

[58] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD*, pages 109–116, 1988.

[59] A. Phanishayee, D. G. Andersen, H. Pucha, A. Povzner, and W. Belluomini. Flex-KV: Enabling high-performance and flexible KV systems. In *Workshop on Management of Big Data Systems (MBDS)*, pages 19–24, 2012.

[60] C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web applications. In *Proceedings of the International Middleware Conference*, pages 155–174, 2004.

[61] D. R. K. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *USENIX NSDI*, pages 43–57, 2015.

[62] D. R. K. Ports and J. Nelson. When should the network be the computer? In *ACM HotOS Workshop*, 2019.

[63] B. Reed and F. P. Junqueira. A simple totally ordered broadcast protocol. In *ACM Large-Scale Distributed Systems and Middleware*, page 2, 2008.

[64] U. Röhm, K. Böhm, H.-J. Schek, and H. Schuldt. FAS — a freshness-sensitive coordination middleware for a cluster of OLAP components. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB '02)*, pages 754–765, 2002.

[65] A. Sapio, M. Canini, C. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. K. Ports, and P. Richtárik. Scaling distributed machine learning with in-network aggregation. *CoRR*, abs/1903.06701, 2019.

[66] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[67] N. K. Sharma, A. Kaufmann, T. E. Anderson, A. Krishnamurthy, J. Nelson, and S. Peter. Evaluating the power of flexible packet processing for network resource allocation. In *USENIX NSDI*, pages 67–82, 2017.

[68] M. Stonebraker and A. Weisberg. The VoltDB main memory DBMS. *IEEE Data Engineering Bulletin*, 36(2):21–27, 2013.

[69] J. Terrace and M. J. Freedman. Object storage on CRAQ: High-throughput chain replication for read-mostly workloads. In *USENIX ATC*, pages 11–11, 2009.

[70] R. Van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *USENIX OSDI*, pages 91–104, 2004.

[71] V. Venkataramani, Z. Amsden, N. Bronson, G. Cabrera III, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, J. Hoon, S. Kulkarni, N. Lawrence, M. Marchukov, D. Petrov, and L. Puzar. TAO: How Facebook serves the social graph. In *ACM SIGMOD*, pages 791–792, 2012.

[72] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *USENIX OSDI*, pages 307–320, 2006.

[73] P. Xiong, H. Hacigumus, and J. F. Naughton. A software-defined networking based approach for performance management of analytical queries on distributed data stores. In *ACM SIGMOD*, pages 955–966, 2014.

[74] H. Zhu, Z. Bai, J. Li, E. Michael, D. Ports, I. Stoica, and X. Jin. Harmonia: Near-linear scalability for replicated storage with in-network conflict detection. In *Technical Report (https://arxiv.org/abs/1904.08964)*, 2019.

[75] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore. NetFPGA SUME: Toward 100 Gbps as research commodity. *IEEE Micro*, 34(5):32–41, 2014.