In-Network Leaderless Replication for Distributed Data Stores

Gyuyeong Kim
Korea University
Seoul, South Korea
gykim08@korea.ac.kr

Wonjun Lee
Korea University
Seoul, South Korea
wlee@korea.ac.kr

ABSTRACT

Leaderless replication allows any replica to handle any type of request to achieve read scalability and high availability for distributed data stores. However, this entails burdensome coordination overhead of replication protocols, degrading write throughput. In addition, the data store still requires coordination for membership changes, making it hard to resolve server failures quickly. To this end, we present NetLR, a replicated data store architecture that supports high performance, fault tolerance, and linearizability simultaneously. The key idea of NetLR is moving the entire replication functions into the network by leveraging the switch as an on-path in-network replication orchestrator. Specifically, NetLR performs consistency-aware read scheduling, high-performance write coordination, and active fault adaptation in the network switch. Our in-network replication eliminates inter-replica coordination for writes and membership changes, providing high write performance and fast failure handling. NetLR can be implemented using programmable switches at a line rate with only 5.68% of additional memory usage. We implement a prototype of NetLR on an Intel Tofino switch and conduct extensive testbed experiments. Our evaluation results show that NetLR is the only solution that achieves high throughput and low latency and is robust to server failures.

PVLDB Reference Format:
Gyuyeong Kim and Wonjun Lee. In-Network Leaderless Replication for Distributed Data Stores . PVLDB, 15(7): 1337 - 1349, 2022.
doi:10.14778/3523210.3523213

1 INTRODUCTION

Data-intensive applications like recommender systems [33] and web search [35] rely on distributed data stores [15], which are supported by non-relational key-value databases like Redis [12], Memcached [25, 29], and RocksDB [5]. The applications often require hundreds of thousands of data store accesses to process sequential user requests. Workloads are typically read-heavy, but write-heavy or mixed ones are also common [11, 38]. To achieve good user experience, data stores should provide high performance for reads and writes, fault tolerance, and strong consistency (i.e., linearizability [16]).

Replication is a common technique in distributed data stores to mask failures and improve availability [8]. Data stores typically employ leader-based protocols, which consist of a leader for handling requests and multiple followers for availability [9, 20, 32]. Handling requests through a leader makes it easy to ensure linearizability, but the leader becomes the performance bottleneck of the data store. It also causes downtime until server failures are resolved because of consensus issues like leader election and group membership changes [17, 22, 30, 37].

Leaderless replication [1, 13, 21, 28] is an approach that allows any replica to process reads and writes for scalable read performance and high availability. Unfortunately, this degrades write performance by increasing the coordination overhead of replication protocols. For example, Chain Replication (CR) [32], a typical leader-based protocol, requires only two messages for a write at the bottleneck replica. However, in Hermes [21], the state-of-the-art leaderless protocol, write coordinator replicas process $3n-1$ messages for a write where n is the number of replicas. Furthermore, although leaderless replication requires no consensus for leadership, it still needs coordination for membership changes. Therefore, when a server fails, request processing stalls until group membership reconfiguration and the corresponding coordination are finished. The aforementioned limitations of leaderless replication motivate us to ask the following question: *can we build a replicated data store that supports high performance, fault tolerance, and linearizability simultaneously?*

To answer the above question affirmatively, this paper presents in-Network Leaderless Replication (NetLR), a replicated data store architecture that leverages switches as an in-network replication orchestrator. Harmonia [40] is a recent effort that utilizes switches for replication, but it is limited to read-write conflict detection for leader-based replication. NetLR goes further than Harmonia by *moving the entire replication functions into the network for leaderless replication.* The key insight behind NetLR is that the switch is an attractive vantage point to perform replication functions, including consistency-aware read scheduling, write coordination, and active fault adaptation. This is because 1) the network switch is a centralized point that provides a global view for replication messages; 2) programmable switch ASICs like Intel Tofino [6] have enough flexibility and computational capability to process replication messages with custom metadata. Compared to replication protocols, our network-level approach enables the data store to enjoy scalable reads, high availability, and linearizability without write performance degradation caused by inter-replica coordination. Furthermore, since the switch directly coordinates writes without write coordinator replicas, latency is also improved. Lastly, the data store does not have to coordinate for membership changes, enabling seamless fault adaptation.

In NetLR, the switch data plane consists of the consistency-aware read scheduling module and the write coordination module. The

Proceedings of the VLDB Endowment, Vol. 15, No. 7 ISSN 2150-8097.
doi:10.14778/3523210.3523213

consistency-aware read scheduling module can preserve linearizability in a leaderless manner even if read-write conflicts occur. Concretely, the module tracks the latest known consistent replica for temporarily inconsistent objects during write operations. Based on this, the switch can forward the read request to the consistent replica even with ongoing writes. The write coordination module initiates and commits write operations. To provide a strict ordering for writes, the module serializes writes by assigning a global sequence number for every write. For high performance, the module makes copies of a write request inside the switch, not receiving from clients. The switch broadcasts writes to every replica, and commits the write only if every reply is aggregated to the switch.

Our switch control plane handles server failures quickly with the active fault adaptation module. Specifically, the switch control plane actively manages the liveness of servers in a centralized manner. When failures occur, the module immediately excludes the faulty replica from the replica list and updates the switch data plane. Since membership reconfiguration is performed without the coordination of servers, requests can be forwarded to live replicas seamlessly. This also implies that clients and storage servers do not have to maintain the liveness state of each other.

NetLR is a practical solution, which can be deployed with commodity switches. We have implemented a prototype of NetLR with Intel Tofino switch ASICs [6] in P4 [10]. NetLR uses only 5.68% of the switch memory because our data plane stores only object metadata. Our solution does not harm packet forwarding functionality and preserves the line rate because the modules can be implemented using a few match-action tables and stateful registers.

To evaluate NetLR, we built a testbed consisting of 7 servers connected to the Edgecore Wedge100BF-32X switch and conduct a series of experiments. The key results are as follows. First, NetLR provides higher throughput and lower latency than existing replication protocols. Furthermore, NetLR supports near-linear scalability. NetLR also maintains high throughput even in the presence of server failures. Lastly, NetLR is robust to workload dynamics like write ratios and object access patterns.

In summary, we make the following contributions.

- We present NetLR, a replicated data store architecture that provides high performance, fault tolerance, and linearizability by performing the entire replication functions in the network switch, eliminating the coordination overhead of replication protocols.
- We design the switch data plane and control plane modules that perform on-path in-network replication while consuming only a small portion of switch hardware resources, including memory and stateful registers.
- We implement a prototype of NetLR and conduct experiments to show that NetLR provides better performance than existing works and is robust to workload dynamics.

The remainder of the paper is organized as follows. In Section 2, we describe the motivation of this work. Section 3, Section 4, and Section 5 provides the design rationale, the architecture, and design of NetLR, respectively. We present implementation and evaluation results in Section 6 and Section 7, respectively. We discuss related work in Section 8. Lastly, we conclude our work in Section 9.

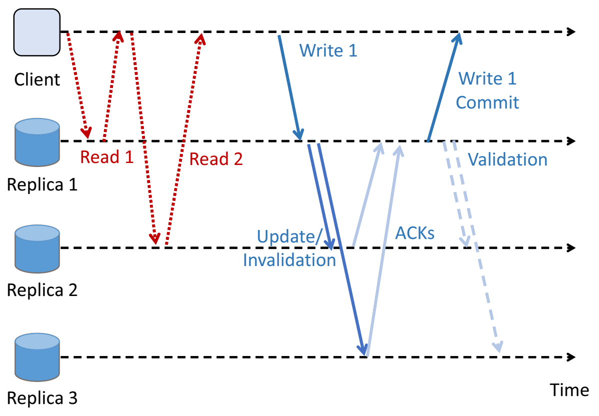


Figure 1: Request processing with Hermes [21], the state-of-the-art leaderless replication protocol. While reads are processed locally, writes require inter-replica coordination.

2 BACKGROUND AND MOTIVATION

2.1 Linearizable Replication Protocols

Distributed data stores with non-relational key-value databases like Redis [12] and RocksDB [5] are the key building block for modern data-intensive applications. Object data is generally replicated over multiple storage servers (i.e., replicas) to improve availability. Since the number of replicas and the deployment cost are in proportion, 3 to 7 replicas are commonly used [17, 21]. To handle requests, a replicated data store uses replication protocols as follows.

2.1.1 Leader-based Replication. Replication protocols are generally leader-based, which consists of a leader replica and multiple follower replicas. In Primary-Backup (PB) [9], the leader replica handles both read and write requests. The follower replicas exist for only backup. The leader in PB initiates a write by broadcasting write update messages to the followers, and commits the write only if all acknowledgments are received. One of the follower replicas is elected as a new leader by consensus protocols like Paxos [22, 30, 31, 39] when the leader fails.

Chain Replication (CR) [32] is a popular leader-based replication protocol. CR improves the overall performance of PB by using different dedicated (leader) replicas for reads and writes in a replica chain. Reads are handled by the tail replica, and writes are initiated by the head replica and forwarded to the tail replica through the chain. The tail replica sends write reply messages to the client to commit the operation. The leader in PB should process $2n$ messages for a single write where n is the number of replicas. However, CR needs to process only two messages at the tail replica thanks to serialized write propagation.

2.1.2 Leaderless Replication. Recently, leaderless replication has gained popularity [1, 13, 21, 27, 28]. In leaderless replication, there is no leader and every replica can serve reads and writes. This offers scalable read throughput because reads can be served locally by multiple replicas[1]. The data store also becomes highly available because the system does not rely on a specific replica for handling requests. Naturally, it requires no consensus for leadership.

Figure 1 shows examples of request processing with Hermes, which is the state-of-the-art leaderless replication protocol [21]. Client requests can be forwarded to any replica. Upon receiving a read request, the replica returns the data of the requested object locally. We can see that Read 1 and Read 2 are processed locally without coordination with the other replicas. Unlike reads, processing writes requires the coordination between the replicas because every replica should maintain the latest data. Therefore, when a replica receives a write request, the replica becomes a write coordinator (Replica 1 in the figure), and broadcasts messages to the other replicas for write updates and object invalidation. After receiving all the update acknowledgments, the write coordinator sends back the write completion message to commit the operation. Lastly, the coordinator propagates object validation messages to enable reads on the object again.

2.2 Inter-Replica Coordination Overhead

As a trade-off for read scalability and high availability, leaderless replication increases the coordination overhead of replication protocols. In addition, the coordination for membership changes is still required. This makes it hard to adapt to server failures quickly.

2.2.1 Coordination for Writes. A pitfall of leaderless replication is that it is challenging to guarantee linearizability. To be linearizable, a read should always get the newest data. This also imposes that writes should be applied in order. However, since requests are processed in a distributed manner, concurrent requests can conflict at a time. This leads to the violation of linearizability as follows.

Read-write conflicts occur when a read accesses a replica where an ongoing write for the same object is not applied yet. In this case, although there exists at least one replica having the latest data in the data store, the read gets stale data. This is because 1) the client does not know which replica has the latest data; 2) it takes time for the write coordinator to update all the replicas by propagating write update messages (i.e., replication lag). Inter-write conflicts occur when multiple replicas coordinate different writes for the same object. Replicas update the object data in the order they receive messages. Therefore, when the update message of the older write arrives at a replica later than that of the newer write, the replica eventually stores the older object data.

Existing leaderless replication protocols sacrifice performance for linearizability by performing extra coordination. In Hermes, each replica maintains the object states, which can be either valid or invalid. As shown in Figure 1, the write coordinator not only updates the object data but also invalidates the object in the other replicas. It also sends back additional messages to validate objects after committing the write. The object invalidation prevents reads from obtaining the stale data but reads stall until the object state becomes valid again. Furthermore, the write coordinator in Hermes should process $3n - 1$ messages for a write coordination, whereas CR only processes two messages at the tail replica for a write. Note that n is the number of replicas. Concretely, the write coordinator in Hermes should process $2n$ messages to initiate and commit a write and $n - 1$ messages to validate objects, respectively. This means that the write throughput in Hermes decreases as the number of replicas grows, while CR provides a constant throughput.

[1]Write throughput cannot be scalable because a write should be applied in every replica. Therefore, the maximum write throughput is limited to the throughput of the single storage replica.

2.2.2 Coordination for Membership Changes. Not surprisingly, in leaderless replication, replicas do not have to reach a consensus for leadership. This is a huge benefit for replicated data store because reaching consensus is a complex problem [20, 22, 30]. However, replicas still need to coordinate for membership changes. Clients and replicas in the data store should maintain the correct liveness state of the other replicas. If the state is incorrect, reads can be delivered to the faulty replica. Writes also cannot be committed due to waiting for the acknowledgment permanently. Furthermore, ideally, the data store should be able to adapt to failures quickly as much as possible. However, when server failures occur, throughput can be degraded until updated membership states are propagated.

3 DESIGN RATIONALE

A case for in-network replication. Our goal is *to build a replicated data store that provides high performance, fault tolerance, and linearizability simultaneously.* Specifically, a solution should provide high throughput and low latency for both reads and writes. To achieve fault tolerance, the data store should seamlessly adapt to server failures. For linearizability, reads must always get the newest data of requested objects and writes must be applied in order.

To achieve the goal, instead of designing a replication protocol, we move the entire replication functions into the network by utilizing switches as an in-network replication orchestrator. The ToR (Top-of-Rack) switch is an attractive vantage point to coordinate replication because every replication message in the storage rack passes through. This indicates that the switch can perform on-path replication with a global view for messages being exchanged between the replicas. The root cause of the inter-replica coordination overhead is that object and server states are maintained in a distributed manner. Therefore, we can eliminate the coordination overhead by maintaining the states at the switch.

The flexibility and capability of programmable switches are the key enablers that realize the in-network leaderless replication. Traditional switches provide only fixed functions like L2/L3 packet forwarding, and we cannot program switch hardware. However, emerging programmable switch ASICs like Intel Tofino [6] and Cavium Xpliant [2] allow us to customize the packet processing pipeline while providing Tbps-scale throughput. Specifically, 1) we can program the packet parser to identify custom packet formats; 2) we can utilize the stateful memory to read/write custom data, and 3) we can define custom match tables and the corresponding actions. In the context of replication, we can make the switch identify replication messages, maintain object and server states, and perform replication functions.

Limited resource capacity of network switches. Conceptually simple, transforming the switch to the replication orchestrator is not straightforward because switch hardware is basically designed for packet forwarding. Programmable switches have limited computational and storage capacity that can be utilized for

Table 1: Comparison to existing works. (coordi. = coordination; mem. = Membership changes; n = number of replicas).

	CR [32]	Hermes [21]	Harmonia [40]	NetLR
No leader election	×	√	×	√
Read scalability	×	√	√	√
No coordi. for writes	×	×	√	√
No coordi. for mem.	×	×	×	√
# hops for a write	$n+1$	4	$n+1$	2
# msgs for a write	2	$3n-1$	2	2

custom functions. Therefore, when designing NetLR, we carefully consider these resource limits. NetLR is a light-weight solution that consumes a little switch hardware resources. Furthermore, NetLR can finish request processing within the available computation budget supported in the switch architecture.

Comparison to Harmonia. Harmonia [40] is the recent effort that exploits switches for replicated data stores. Specifically, Harmonia detects read-write conflicts in the switch. We highlight the distinct features of NetLR as follows.

- Harmonia only supports leader-based replication protocols (e.g., PB and CR). The switch forwards reads to the leader replica if the requested object is inconsistent. NetLR performs leaderless replication in the switch, hence no complex replication protocols are needed. If there exist pending writes for the object, the switch forwards reads to the latest known consistent replica, which can be any replica, not a specific one like the leader.
- In Harmonia, the leader replica is in charge of write coordination. This indicates that Harmonia cannot achieve high throughput and low latency for writes at the same time. For example, when Harmonia uses CR, latency increases as the chain length increases. Unlike Harmonia, the latency in NetLR is not affected by the number of replicas because we coordinate writes using the switch in a broadcasting manner.
- Harmonia handles membership changes passively based on replication protocols. NetLR actively deals with group membership changes. The switch control plane monitors the liveness of replicas and adapts to server failures quickly without the intervention of replicas.

Comparison to existing works. We summarize the difference of NetLR compared to existing works in Table 1. CR and Hermes are protocol-level approaches, whereas Harmonia and NetLR are network-level solutions. Hermes and NetLR provide read scalability and require no leader election since they are leaderless. Harmonia provides read scalability as well with in-network read-write conflict detection. NetLR is the only solution that requires no inter-replica coordination for writes and membership changes. There exist no write coordinator among replicas. Therefore, our solution requires only two messages (i.e., write update and acknowledgment) to be processed at the bottleneck replica for a write operation. In the same vein, two hops are enough to handle a write (i.e., 1 Round Trip Time (RTT)).

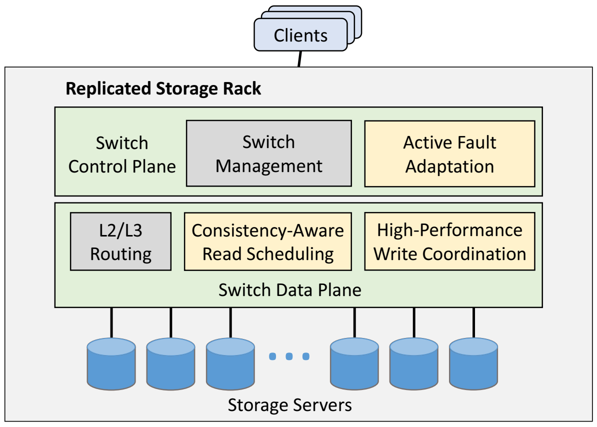


Figure 2: NetLR architecture.

4 NETLR ARCHITECTURE

Figure 2 illustrates the NetLR architecture that consists of the following components.

4.1 Switch Data Plane

The switch data plane is the core component of NetLR that orchestrates the replication process. The packet processing pipeline in the switch data plane basically performs L2/L3 packet forwarding as a general switch. NetLR adds the replication functions to the processing pipeline, and this does not harm the packet forwarding functionality. The functionality of NetLR is implemented with the following two modules.

4.1.1 *Consistency-aware Read Scheduling.* The consistency-aware read scheduling module schedules read requests by resolving read-write conflicts in a leaderless manner. To do this, the module maintains the list of objects that become inconsistent due to ongoing writes. The list is generally small because the object is removed from the list when the switch commits the write. The module also tracks the ID of the latest known consistent replica having the latest data of inconsistent objects. When the requested object is in the inconsistent object list, the switch forwards reads to the latest known consistent replica. This means that reads always get the latest data for an object, even if a write for the object is not applied to all the replicas yet. The replica ID is updated to that of the source replica when a new write reply arrives at the switch. This enables the switch to balance read requests even when the object is inconsistent, unlike Harmonia that forwards reads to the leader replica only.

4.1.2 *High-Performance Write Coordination.* The write coordination module in the switch directly coordinates writes. Therefore, replicas do not communicate with each other to coordinate writes. Upon receiving writes from clients, the switch module clones write messages and broadcasts them to the replicas. When every write reply is aggregated to the switch, the module commits the write operation by sending the write reply to the client. This module greatly reduces the required number of messages for write coordination. Specifically, since no inter-replica coordination is required,

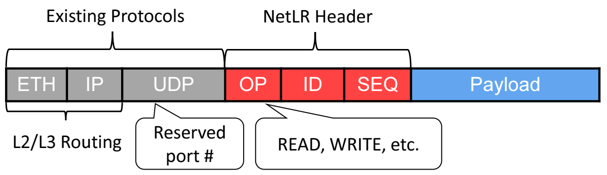


Figure 3: NetLR packet format.

a replica processes only two messages for a single write. Thus, we can see a constant write throughput regardless of the number of replicas.

The module also resolves inter-write conflicts by maintaining a global sequence number that monotonically increases upon receiving every write. The switch assigns the sequence number to writes, and this enables the replicas to distinguish the order of write requests, including the concurrent writes for the same object. We make the storage replicas accept the write only if the sequence number in the message is larger than or equal to the maintained sequence number. Therefore, we do not have to invalidate the object to avoid out-of-order write updates when different writes for the same object are processed.

4.2 Switch Control Plane and Servers/Clients

4.2.1 Active Fault Adaptation. The active fault adaptation module in the switch control plane maintains the liveness state of replicas in addition to relevant information (e.g., IP address, the total number of replicas, and replica ID). The module periodically ($\approx 10ms$) updates the liveness state by checking the port status connected to the replicas. When a port down (i.e., a server failure) is detected, the module immediately excludes the faulty replica from the replica IP and port number lists. The module also updates registers and match-action rules in the switch data plane, which include the number of live replicas and the translation tables for replica IDs and IP addresses. No actions are required by replicas to handle failures. Thanks to the module, the data store can adapt to server failures quickly without relying on replication protocols.

4.2.2 Storage Servers and Clients. NetLR greatly simplifies replication protocols because the entire replication functions are performed in the switch. When read requests arrive, the storage server (i.e., replica) sends read replies with the data of the requested object. Upon receiving write requests, the server updates the object data and sends back write replies only if the sequence number of the packet is larger than or equal to the maintained sequence number. Clients just send requests and receive replies. We note that NetLR does not make local requests go through the switch. The network stack of a server forwards local requests to the server itself via a loopback network interface.

5 NETLR DESIGN

5.1 Packet Format

Figure 3 shows the packet format of NetLR. NetLR uses a custom L7 protocol message. We reserve an UDP port number for NetLR so that the switch can apply different packet header parser flows

Algorithm 1 Request Processing in Switch

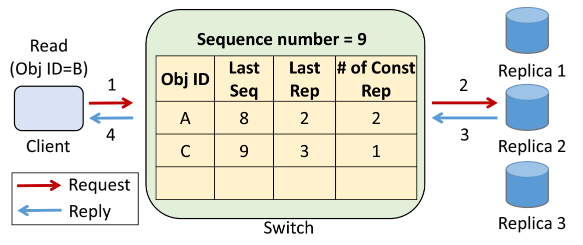
```
    – pkt: Packet to be processed
    – Obj: List of inconsistent objects
    – CRep: List of the latest known consistent replicas
    – LSeq: List of the last written sequence number
    – seq: Sequence number for writes
1:  if $pkt.op$ == READ then
2:      if $pkt.id \in Obj$ then                ▷ Read for inconsistent object
3:          $pkt.dst \leftarrow CRep[pkt.id]$
4:      else                                    ▷ Read for consistent object
5:          $pkt.dst \leftarrow$ choose a replica using schedulers
6:      end if
7:      Forward($pkt$)
8:  else if $pkt.op$ == WRITE then
9:      $seq \leftarrow seq + 1$                ▷ Increase sequence number for every write
10:     $pkt.seq \leftarrow seq$                ▷ Assign sequence number
11:     $Obj.Insert(pkt.id)$                    ▷ Add object to list
12:     CloneForward($pkt$)                     ▷ Broadcast writes
13: end if
```

Algorithm 2 Reply Processing in Switch

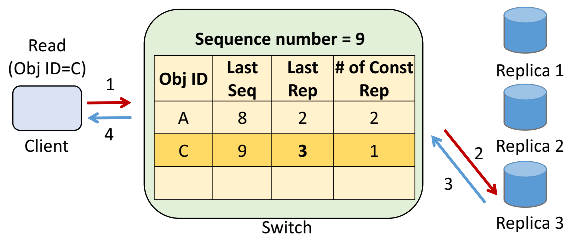
```
    – NumCRep: List of the number of consistent replicas
    – RepID: List of replica IDs
1:  if $pkt.op$ == R-REPLY then
2:      Forward($pkt$)                          ▷ No specific action for read reply
3:  else if $pkt.op$ == W-REPLY then
4:      if $pkt.seq > LSeq[pkt.id]$ then
5:          $LSeq[pkt.id] \leftarrow pkt.seq$   ▷ Update the last seq. number
6:          $NumCRep[pkt.id] \leftarrow 1$      ▷ Reset for new write
7:          $CRep[pkt.id] \leftarrow RepID[pkt.src]$
8:      else if $pkt.seq == Lseq[pkt.id]$ then
9:          $NumCRep[pkt.id] \leftarrow NumCRep[pkt.id] + 1$
10:         $CRep[pkt.id] \leftarrow RepID[pkt.src]$
11:     else if $pkt.seq < Lseq[pkt.id]$ then
12:         Drop($pkt$)                         ▷ Discard older write
13:     end if
14:     if $NumCRep[pkt.id]$ == $\#ofReplicas$ then
15:         $Obj$.Remove($pkt.id$)              ▷ Remove object
16:         Forward($pkt$)                      ▷ Commit the write
17:     else
18:         Drop($pkt$)                         ▷ Not enough to commit
19:     end if
20: end if
```

for NetLR packets and normal packets. The NetLR message has the header consists of three fields as follows.

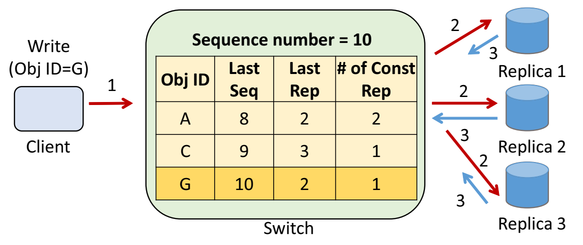
- OP: the message operation type, which can be READ, WRITE, R-REPLY, and W-REPLY.
- ID: the ID of a requested object.
- SEQ: the sequence number for write requests. The switch generates a value for every write request.



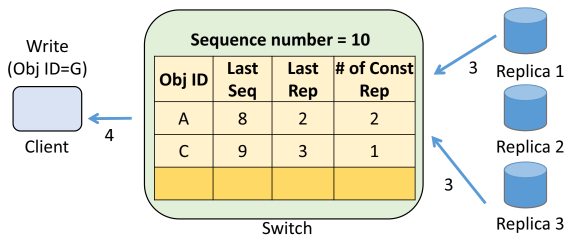
(a) Read for consistent objects



(b) Read for inconsistent objects



(c) Write propagation



(d) Write completion

Figure 4: Request and reply processing in NetLR. The switch can forward reads to any replica for consistent objects. If inconsistent, reads are forwarded to the latest known consistent replica only. The switch coordinates writes by tracking inconsistent object states in the data plane. (Last Seq = last written sequence number; Last Rep = latest known consistent replica ID; # of Const Rep = number of consistent replicas).

5.2 Request and Reply Processing

NetLR has a different packet processing logic depending on the type of message as follows. Algorithm 1 is the pseudocode of request processing in the switch.

5.2.1 Read Requests. The switch in NetLR always forwards reads to replicas having the latest data, ensuring linearizability. When a read request is received, the switch first checks whether the requested object is in the inconsistent object list (lines 1-2). If the object is included in the list, the switch updates the destination of the packet to the latest known consistent replica (lines 2-3). Otherwise, the switch can select any replica as the destination using request schedulers (e.g., round-robin in our work), since the object is fully consistent (lines 4-5). After that, the switch forwards the read to the destination replica (line 7).

5.2.2 Write Requests. The NetLR switch provides in-order writes and high performance through in-switch write serialization and write coordination. Upon receiving a write, the switch increases the global sequence number and assigns the number to the write (lines 8-10). Next, the switch inserts the object into the inconsistent object list (line 11). After that, the switch propagates writes using packet cloning (line 12). Specifically, the request is copied as many as the number of replicas, and each request is sent to each replica.

5.2.3 Read and Write Replies. Algorithm 2 is the pseudocode of reply processing in the switch. Since reads can be performed locally using a single replica, the switch just forwards read replies to the client (lines 1-2). However, for write replies, the switch performs additional actions.

If the sequence number of the packet is larger than the last written sequence number, this reply is of the newer write (lines 3-4). The switch updates the last written sequence number for the object to the sequence number of the packet (line 5). Next, the switch resets the number of consistent replicas to 1 because a new write coordination begins (line 6). Lastly, the latest known consistent replica is updated to the ID of the source replica that sent the reply (line 7). If the reply is for the current write, the switch first increases the number of consistent replica for the object by 1 (lines 8-9). As the same as the newer write, the latest known consistent replica is also updated (line 10). Upon receiving a reply of older writes, the switch simply drops the packet since we do not need the message anymore (lines 11-12).

If the number of consistent replica is equal to the number of replicas, this means that all the replicas successfully updated the object data (line 14). The switch now removes the object from the inconsistent object list, since the object is consistent (line 15). The switch finally commits the write operation by forwarding the reply to the client (line 16). Otherwise, the packet is dropped because we need more consistent replicas to commit the write (lines 17-18).

5.2.4 Operational Examples. We now show examples of request and reply processing in NetLR. Figure 4 (a) illustrates the processing of a read for consistent object B. The switch can choose any replica as the destination since all the replicas contain the newest data for the object. In this example, the switch forwards the request to replica 2. The replica simply returns the reply with the object data to the client. Figure 4 (b) is the example of a read for inconsistent object C. Unlike the example in Figure 4 (a), the switch can forward the request to the latest known consistent replica only (replica 3 in the example). The replica finishes the operation by returning the reply to the client.

Figure 4 (c) shows the example of a write for object G. Upon receiving the write request, the switch increases the global sequence number by 1 and assigns the number to the request. Therefore, the sequence number is updated to 10. A write should be applied in all the replicas, hence the switch clones the write and broadcasts the messages. At the same time, the object G is inserted in the inconsistent object list. Each replica returns the reply after updating the data of object G. In the example, between the replicas, the reply of replica 2 arrives at the switch first. The switch updates the last written sequence number, the latest known consistent replica, and the number of consistent replicas, to 10, replica 2, and 1, respectively. Figure 4 (d) shows how a write for object G is committed. The write replies from replica 1 and replica 3 finally arrive at the switch. The switch also updates the object states for each reply. Since every replica is acknowledged, the switch removes object G from the list. In addition, the switch forwards the reply of the latest replica (i.e., replica 3) to the client to commit the write.

5.3 Handling Failures

5.3.1 Dropped Messages. Since the network is unreliable, messages can be dropped. If a write request sent from the switch or a write reply is dropped, an object remains in the inconsistent object list permanently. In this case, reads are forwarded to the latest known consistent replica until a newer write is performed for the object. This may degrade performance for a while but does not harm consistency. We can also use application-level retransmission mechanisms like a timeout to prevent the client from waiting for the write commit message excessively when a write reply is dropped. The timeout can be adjusted dynamically by considering the latency trends to avoid early or long timeouts.

5.3.2 Server Failures. In NetLR, server failures are handled by the active fault adaptation module in the switch control plane as described in Section 4.2.1. The addition or recovery of replicas can be handled by relaunching the switch control plane again after finishing reconfiguration and copying data. Specifically, we first copy object data of an up-to-date replica to the recovered replica. We also make the up-to-date replica forwards copied write requests to the added replica until the serving request of the added replica is resumed. Lastly, we update the membership information in the control plane and resume serving requests of the recovered replica.

5.3.3 Switch Failures. Switches have the lowest failure rates across all data center hardware. For switches that experience at least one failure, the mean number of failures for a year is only 1.1 [14]. When switch failures occur, we can reboot the switch or replace it with a backup switch. This can affect availability, performance, and consistency. Specifically, during downtime, the data store becomes unavailable, and the performance is also degraded. The time to restart the switch depends on switch hardware. In our experience, it takes tens of seconds to boot and run the switch process.

The availability and performance issues are not specific to NetLR because the switch failure impacts any type of distributed system. The key issue related to NetLR is consistency, since the switch loses the maintained object states. For example, replicas may drop write requests until the sequence number of the switch reaches to that of the replica because the switch failure resets the global sequence number. To avoid this, as in existing works [23, 40] we can use the switch's unique ID that is monotonically increased when the switch starts. By comparing the switch's ID in addition to the sequence number, the server can accept the writes of the new switch. The switch ID also can be used to prevent inconsistent reads, which are caused by lost object states. Specifically, we can insert the switch ID into the read requests. Storage replicas accept reads only if the carried switch ID is the same as the latest known switch ID. This makes read requests being dropped until a new write is performed for the object, but linearizability can be preserved. To mitigate the performance degradation caused by the switch ID-based solution, we may use a snapshot of the data store instead of the switch ID. This can preserve consistency and does not harm performance, although the object state turns back to the time when the snapshot is created.

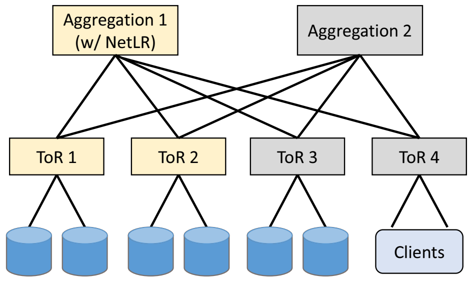


Figure 5: The aggregation switch as the in-network replication orchestrator for multi-rack deployment.

5.4 Discussion

5.4.1 Multi-Rack Deployment. As illustrated in Figure 2, we basically consider a storage rack for deployment where all replicas are co-located in the same rack. This single-rack deployment model can be applied to many practical use cases, such as on-premise data stores for enterprises and specialized data stores for the cloud data center [40].

However, some data stores may use replicas in different storage racks for replication. In this case, since the object states are distributed in multiple ToR switches, the functionality of NetLR may work incorrectly. To address this, we can utilize the aggregation switch that interconnects multiple storage racks as shown in Figure 5. In the figure, we can see that the aggregation 1 switch coordinates replicas under ToR 1 and ToR 2 switches. By forwarding requests of a replica group to the same aggregation switch, we can monitor every replication message from multiple racks. This does not increase the latency unnecessarily unless the client was in the same storage rack of the destination replica because every packet from the client passes through the aggregation switch to be forwarded to the destination replica. The control plane of the aggregation switch also can monitor the port status of connected ToR switches because the switch control planes also can communicate with each other. However, it may increase the time to adapt server failures due to increased network hops. The correct design and evaluation for multi-rack deployment are our future work.

5.4.2 Supporting Large Object Data. NetLR uses UDP because the protocol supports low latency for key-value databases [8, 12, 36]. Most requests and replies are delivered with a single packet because most object values are tens of bytes [29]. This is less than the Maximum Transmission Unit (MTU) size that is typically 1500-byte. However, some workloads may have larger values, which are fragmented into multiple packets. In this case, the switch can perceive the packets of a single write as the packets of multiple writes for different objects. To avoid this, like Harmonia [40], we can differentiate the processing of the first packet from the following packets using different operation types. Specifically, the switch forwards the following packets without accessing the hash table, avoiding duplicate packet processing.

5.4.3 Replication to A Subset of Replicas. The current NetLR design targets replicated data stores where objects are replicated over all replicas connected to the switch. However, data stores may desire to replicate objects in only a subset of the replicas. NetLR can support this by adding the replica group ID to message metadata. In particular, for write coordination, the switch can use a different multicast group matched to the replica group ID in the request message. For reads, we can leverage a register value that expresses the replicas belonging to the replica group as a bitmap.

6 IMPLEMENTATION

6.1 Switch Control Plane and Servers/Clients

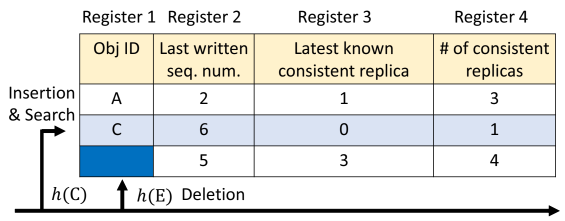
6.1.1 Control Plane. The switch control plane application is written in Python 2.7 using Barefoot Runtime APIs. When we run the application, the switch control plane first updates the switch data plane with pre-configured table rules and register values. After that, the application refers the port status periodically for active fault adaptation.

6.1.2 Client-Server Application. We also make a single-threaded client-server application in Python 3.7.10 with the Redis [12] API for Python. Our application is based on `pypacker` library [4] because it allows us to define and manipulate custom packet headers easily. We also use `pypy3` [3] to maximize packet processing performance of servers. The client measures throughput and latency by communicating with replicas. The server handles requests with replication protocols.

6.2 Data Plane Implementation

It is not straightforward to implement the NetLR design because switch hardware has limited resources and strict timing constraints. Therefore, we use several techniques that translate the design to a working system. Our switch data plane is written in $P4_{16}$ [10] and is compiled with Intel P4 Studio SDE 9.2.0 for Intel Tofino switch ASICs [6]. Overall, NetLR consumes 5 pipeline stages and only 5.68% of the switch memory.

6.2.1 Packet Processing Pipeline. The packet processing pipeline consists of the ingress pipeline and the egress pipeline. Between the two pipelines, the packet buffer exists. Our modules reside in the ingress pipeline since NetLR performs custom packet forwarding for replication messages. The ingress pipeline consists of multiple Match-Action (M-A) stages. In each stage, the switch can perform the corresponding action defined by an M-A table. Normal packets are directly processed by the L2/L3 routing table for packet forwarding. If the packet is of NetLR, the switch applies additional M-A stages to the packet in accordance with the message type. The NetLR pipeline consists of 4 stages, and this does not exceed the budget provided by the switch architecture.



Obj ID	Last written seq. num.	Latest known consistent replica	# of consistent replicas
A	2	1	3
C	6	0	1
	5	3	4

Figure 6: Insertion, search, and deletion in the hash table that tracks the state of inconsistent objects. The index for the registers is the hash of object ID. For deletion, we only remove the object ID to meet the switch computation budget.

6.2.2 Hash Table with Register Arrays. To maintain inconsistent object states, we use multiple register arrays. We use four arrays to store object IDs, the last written sequence numbers, the latest known consistent replica IDs, and the number of consistent replicas. Ideally, we should provide a dedicated register slot for each object. However, the register array size is statically determined at compile time and the available memory space is limited. Therefore, to reduce the memory usage, we utilize the register arrays as the hash table. Specifically, the index of the arrays is the hash of the object ID, which is stored in the ID field at the NetLR header. Since the hash can be duplicated due to hash collisions, we store the original object ID in a register array. Note that the original object ID can be carried in the packet payload and can be parsed by the switch. The hash can be computed in the switch data plane, but we generate the hash from the client to simplify the processing pipeline.

Figure 6 shows the hash table design and how the switch performs operations. For insertion, the switch first reads the matched slot in the object ID array. If the slot is empty, the switch records the object ID in the slot to avoid overwrite of collided objects. For search, the switch simply returns the stored value in the matched slot. To delete the object from the hash table when committing writes, the switch removes the store object ID from the matched slot. One notable point is that we do not remove the values of the other register arrays. This is because the removal of the values from all the register arrays consumes a lot of M-A units, which may exceed the computation budget. To avoid accesses that lead to misbehavior, the other register arrays can be accessed only if the stored object ID is equal to the object ID in the packet.

6.2.3 Handling Hash Collisions. The hash of different objects can collide due to the nature of hash functions. As we have described above, we prevent the access of collided objects by storing the original object ID in hash slots. The switch simply drops the packet of collided objects. This may be retransmitted by the client with an application-level loss recovery mechanism. The performance degradation caused by hash collisions is not heavy because the object state is maintained only until committing the write operation. This also means that there exist a few inconsistent objects at a time. In addition, a typical production workload is read-heavy with 5% of write ratio [21, 26, 40]. We show that 128K hash slots are enough to serve the write-only workload in the performance evaluation, which is the most challenging workload. If necessary, we can also utilize the well-known open addressing technique and double-hash as Harmonia [40] does.

6.2.4 Group Membership States. The switch data plane also maintains the group membership states including the replica ID, the IP address, and the port number. To translate the replica ID to the corresponding IP address and vice versa, we use two translation M-A tables. The table rules are updated by the control plane. The client does not have to know the exact IP address of replicas because the switch determines the destination replica using request schedulers for reads and packet cloning for writes. This enables the switch to handle failures without the intervention of storage servers.

6.2.5 Limitations of Switch Hardware. Our data plane implementation shows that we need careful approaches and techniques to meet the limitations of programmable switch ASICs. However, we clarify that this is a data plane design for a specific switch architecture. Note that each switch architecture may have different advantages and limitations. The design of NetLR may be fully implemented in another switch architecture. The programmable switch ASICs have been improved rapidly, and new features have become available with emerging ASICs. For example, Intel Tofino does not provide the port queue length information in the ingress pipeline, which is critical for congestion-aware packet forwarding. However, the queue length information becomes available in the ingress pipeline with emerging Intel Tofino2 [7].

7 EVALUATION

7.1 Experiment Methodology

7.1.1 Testbed. Our testbed consists of 7 servers connected to an Edgecore Wedge100BF-32X with a 3.2 Tbps Intel Tofino switch ASIC [6]. 6 of the servers are storage replicas with an Intel 6-core CPU and 16GB of memory. The maximum throughput of each storage replica is roughly 18.4KRPS (Requests Per Second) for reads and 17.8KRPS for writes. One of the servers acts as the client with an Intel 6-core CPU and 32GB of memory. The servers are equipped with a Mellanox ConnectX-5 Ethernet NIC. Unlike the other servers, the client server is with a dual-port NIC. To use two clients with a single server, we run two client applications and assign a separate CPU core and NIC port. The maximum throughput of a single client is approximately 54KRPS. Therefore, with two clients, the maximum throughput is roughly 108KRPS. We set the link speed to 40Gbps. The servers run Ubuntu 18.04.3 LTS with Linux kernel 5.4.0-77-generic.

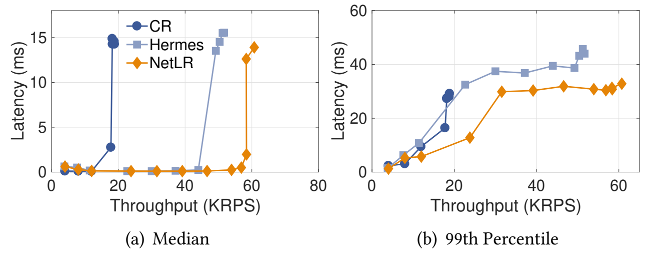


Figure 7: Latency vs. throughput.

7.1.2 Workloads. Unless specified, we basically use four replicas and a typical read-heavy workload of 5% write ratio with uniform distribution like existing works. We use 1M objects with 32-bit IDs and 128-bit values as the same as Harmonia [40].

7.1.3 Compared Schemes. We compare NetLR with CR [32] and Hermes [21], which are typical leader-based and leaderless replication protocols, respectively. Comparison with the protocols lets us know how much in-network replication brings performance gains. Hermes is an RDMA-based protocol using Infiniband NICs. In our experiments, Hermes communicates with replicas using the OS network stack and Ethernet NICs for fair comparison.

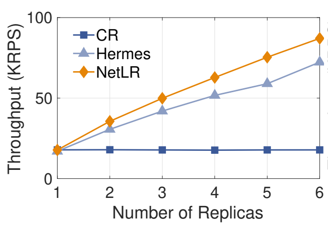
We also compare NetLR against Harmonia [40], the state-of-the-art in-switch solution. Since the code of Harmonia is not publicly available, we have implemented the Harmonia mechanism described in the original paper by doing our best. One difference is that our implementation provides 128K hash slots with a single stage while the original implementation has 192K hash slots with 3 stages. This does not mislead us to incorrect conclusions because 128K slots are enough to handle temporarily inconsistent objects.

The other difference is that our Harmonia implementation does not compare the sequence number to remove the object from the dirty set when processing write completion messages. This may cause the weak consistency for corner cases. In detail, this can cause forwarding of reads to inconsistent replicas if there exist concurrent writes for the same object, because the object is removed from the dirty set by the earlier write completion before the arrival of the newest write completion. To avoid the read stall in the inconsistent replica, we make replicas accept reads without referring to the last committed sequence number. This enables us to see the correct performance by trading consistency. We gracefully note that we compare the performance, not the consistency correctness. Note that NetLR, CR, and Hermes preserve linearizability in our experiments.

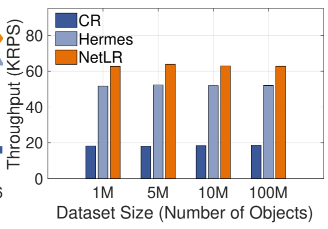
7.2 Experimental Results

7.2.1 Latency vs. Throughput. We first evaluate the latency as a function of the achieved throughput. The clients generate requests to the replicas, and we measure the median latency and the 99th percentile latency (i.e., tail latency) by varying throughput.

We plot the results in Figure 7. It is easy to see that CR achieves the lowest throughput between the compared solutions. This is because CR handles requests with a single replica for each request type. Hermes shows lower throughput compared to NetLR. This is



(a) Impact of number of replicas



(b) Impact of dataset size

Figure 8: Scalability experiments.

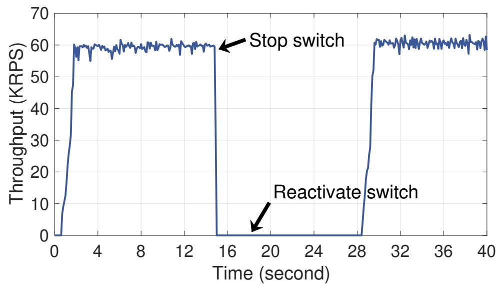


Figure 10: Throughput under switch failures.

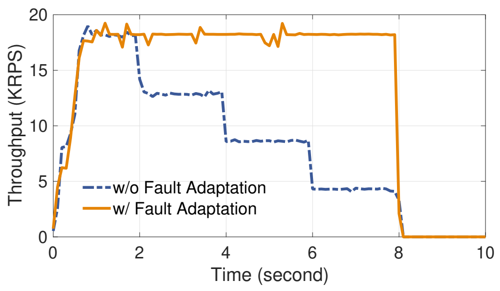


Figure 9: Throughput under server failures.

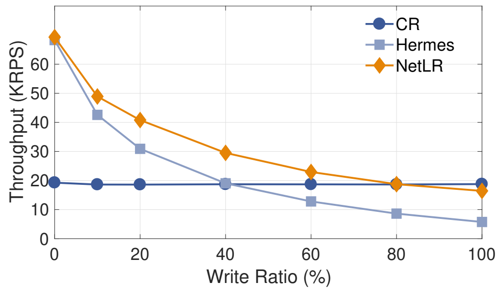


Figure 11: Throughput with different write ratios.

because Hermes uses replicas for write coordination and performs additional coordination to validate objects in replicas.

Figure 7 (a) shows the median latency as a function of throughput. Since the throughput of CR is bounded to the single replica performance, the latency of CR rapidly increases as the throughput reaches the throughput limit of the single replica. The median latency of Hermes also soars when exceeding the saturated throughput. Between the compared schemes, NetLR shows the lowest median latency across the throughput. This implies that NetLR is the solution that provides high throughput and low latency simultaneously for typical workloads. Meanwhile, Figure 7 (b) shows the results for the 99th percentile latency. As the same as the result in the median latency, NetLR offers the lowest tail latency between the schemes. The gap between Hermes and NetLR stems from the fact that Hermes increases the latency of writes by coordinating write operations in replicas, which means each replica should process more messages than NetLR with more RTTs.

7.2.2 Scalability. We evaluate the scalability of NetLR in the number of replicas and the dataset size. For the impact of number of replicas, the clients generate requests with a sending rate of the limit of the client. After that, we measure the throughput by varying the number of replicas. For the impact of dataset size, we vary the dataset size from 1M to 100M with four replicas.

Figure 8 (a) shows the results in the number of replicas. As expected, the throughput of CR is bounded to the single replica throughput. NetLR and Hermes show the increased throughput as the number of replicas increases. However, the throughput of NetLR is higher than Hermes by 1.18× on average. This is because Hermes sacrifices write throughput for inter-replica coordination. Unlike CR and Hermes, we can see that NetLR offers near-linear scalability. Figure 8 (b) shows that the impact of dataset size is trivial. This is because the dataset size itself does not change the required computing resources for read/write operations.

7.2.3 Performance with Failures. In this experiment, we inspect the performance of NetLR under switch and server failures.

For server failures, the clients generate only read requests to four replicas with a sending rate of 18KRPS, which can be fully served by one replica. Every 2 seconds, we cause a failure of one replica by disabling the port. We measure the throughput for 10 seconds with and without the active fault adaptation module. Figure 9 shows the throughput as a function of time. We can see that NetLR maintains the throughput close to the sending rate even with the server failure. This is because our fault adaptation module immediately excludes the faulty replica from the replica list and updates the data plane states. Therefore, the requests are forwarded to the live replicas seamlessly. However, without the fault adaptation module, the switch still forwards reads to the faulty replica, making the read fail.

For switch failures, we manually stop and reactivate the switch. Figure 10 shows the throughput for the period of failure and recovery. We can see that the throughput rapidly decreases as the switch fails. After the switch is reactivated, the throughput is recovered to the same as before the failure. It takes roughly 13 seconds to reactivate switches. We note that this time depends on switch hardware, not the NetLR mechanism.

7.3 Deep Dive

7.3.1 Impact of Write Ratios. We now inspect the impact of write ratios on the system throughput to evaluate the performance for dynamic workloads, including write-heavy and mixed ones. The

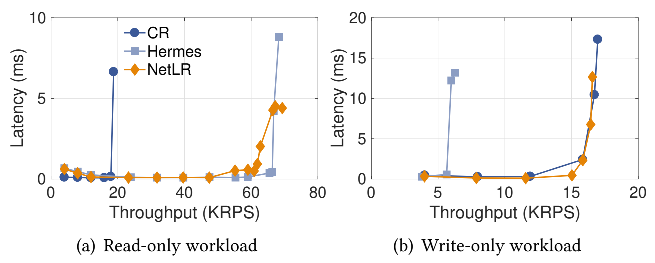


Figure 12: Median latency for extreme workloads.

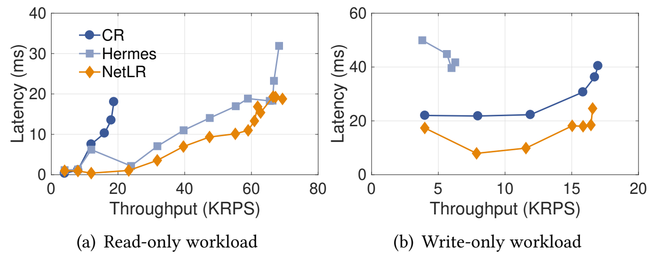


Figure 13: 99th percentile latency for extreme workloads.

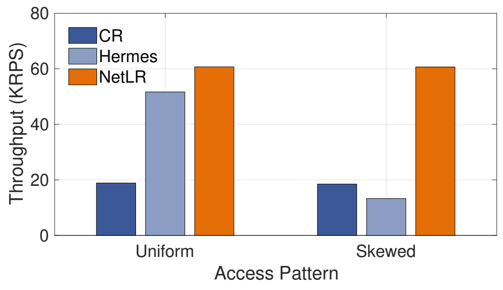


Figure 14: Throughput with different access patterns.

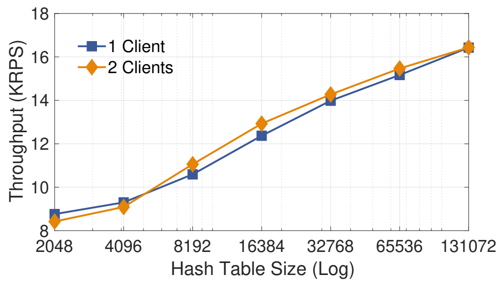


Figure 15: Impact of switch memory size.

client measures the saturated throughput by varying the write ratio for the three compared schemes.

Figure 11 shows the throughput with different write ratios. We can see that CR offers a constant performance regardless of write ratios. The throughput of Hermes decreases as the write ratio grows. This is not surprising, since Hermes performs expensive inter-replica coordination to handle writes. For NetLR, with higher write ratios, the throughput goes close to the performance of CR since the write throughput is bounded to the single replica performance. The throughput of NetLR is slightly lower than CR when the write ratio is 100% due to hash collisions. The result in Figure 11 indicates that NetLR can provide the best throughput for most ratios.

7.3.2 Performance for Extreme Workloads. We conduct a throughput and latency experiment with extreme workloads, which are read-only and write-only. These workloads do not represent typical workloads in production data stores. However, this lets us know the definite performance for each request type.

Figure 12 shows the median latency at different throughput levels for read-only and write-only workloads. The result with the read-only workload is similar to the result in Figure 7 (a) because the gap of write ratio is only 5%. As expected, for the write-only workload, we can see that Hermes results in the lowest throughput due to the inter-replica coordination overhead. The throughput of NetLR is bounded to the single replica performance like CR since a write must be applied in all replicas. Figure 13 shows the 99th percentile latency with different throughput levels for the extreme workloads. We can see that NetLR offers the lowest tail latency between the compared schemes across the workloads thanks to in-network replication. In the write-only workload, the throughput of CR is better than NetLR by 1.08× as already shown in Figure 11. This is because of request drops due to hash collisions in the switch data plane. Although throughput degradation occurs, we gracefully argue that this is not critical since the write-only workload is far from production workloads.

7.3.3 Impact of Access Patterns. We now evaluate the performance of NetLR in skewed workloads. Since we want to see the performance in the worst case, we use the most skewed request distribution. This means that all generated requests are for the same object. This is more challenging than the distribution with zipf-0.9 [21, 40].

Figure 14 plots the throughput of the three schemes with different access patterns. Overall, NetLR provides the best performance in the two workloads. The throughput of NetLR with the uniform distribution is higher than that with the skewed distribution by 1.16×. However, for Hermes, the gap between the distributions is 5.52×. This is because reads frequently stall because of object invalidation by many concurrent writes for the same object. Unlike Hermes, NetLR forwards reads to the latest known consistent replica without request stall. Since requests are serialized through chains, CR shows a constant performance with the both distributions.

7.3.4 Impact of Hash Table Size. We inspect the impact of the switch memory on the performance by varying the hash table size. We use the write-only workload for this experiment. This is because the write performance highly depends on the hash table size because of concurrent writes for the same object. The write-only workload is the most challenging workload that generates many concurrent inconsistent objects. Recall that the write ratio of the typical workload is only 5%.

Figure 15 shows the saturated throughput with different hash table sizes for 1 client and 2 clients cases. Regardless the number of clients, we can see that the throughput grows as the switch provides more hash slots. With 128K hash slots, the throughput

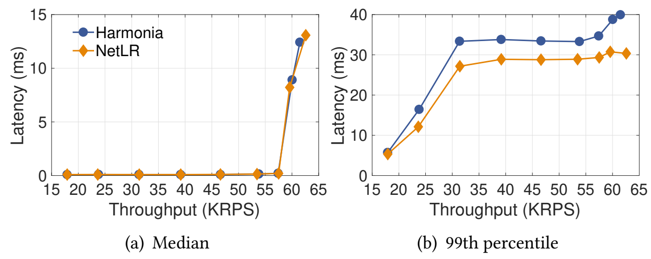


Figure 16: Throughput and latency (vs. Harmonia).

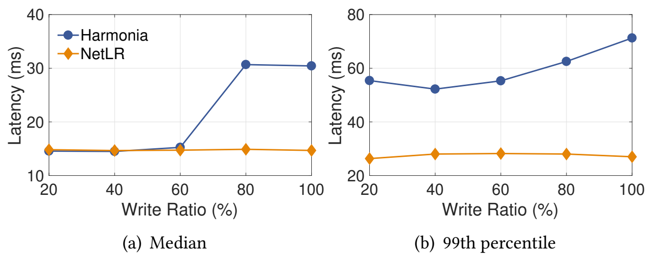


Figure 17: Impact of write ratios (vs. Harmonia).

reaches the maximum throughput of the write-only workload. We also note that a smaller hash table size is enough to serve the typical workload since the workload contains only 5% of write requests. When using 128K hash slots, NetLR requires only 5.68% of the switch memory. By considering that the switch generally provides tens of megabytes of memory, we can say that NetLR requires a small portion of switch resources to track inconsistent objects.

7.3.5 Comparison to Harmonia. We now compare NetLR against Harmonia [40] with the typical workload described in Section 7.1.2. Figure 16 shows the median latency and the 99th percentile latency at different throughput levels. We can see that the two schemes offer the almost same throughput. However, we can see that there are gaps in latency between NetLR and Harmonia. In Figure 17, we find that the latency gap is highly impacted by the write ratio. These results also indicate that NetLR is generally better than Harmonia in tail latency for dynamic workloads.

This is because Harmonia relies on replication protocols (CR in this case) to coordinate writes, whereas NetLR coordinates writes in the switch. In both NetLR and Harmonia, each replica processes two messages to handle a write. However, Harmonia requires longer latency because of write propagation through chaining. One may wonder what if Harmonia uses PB, which is a broadcast-based protocol, instead of CR. However, since PB uses only the single leader replica to coordinate writes, the performance is worse than NetLR because of imbalanced write loads.

8 RELATED WORK

Replication protocols. Replication protocols can be categorized into leader-based and leaderless protocols [8, 21]. PB [9] is the basic protocol that uses the stable leader to handle reads and writes. The other replicas exist for availability only. CR [32] improves the performance of PB using different replicas for reads and writes. CRAQ [34] allows local reads for CR. Mencius [27] and EPaxos [28] are leaderless replication protocols that change the leader replica for different requests opportunistically. Hermes [21] is the state-of-the-art leaderless replication protocol that allows local reads and concurrent write coordination by multiple write coordinators. The major difference of NetLR from the replication protocols is that NetLR is a network-level solution, whereas the others are the protocol-level solutions. Our approach has the advantage over the protocols that enables leaderless replication without performance degradation caused by the inter-replica coordination.

In-network computing. In-network computing is an emerging paradigm that utilizes the network switch for computing acceleration. NetCache [19] offers in-network caching by storing popular object data in the switch. Pegasus [24] improves NetCache by storing popular objects in servers selectively with coordination of the switch. NOPaxos [23] is an early work that utilizes switches for request serialization. The concept of serialization is also used in NetLR as well. Harmonia [40] is the state-of-the-art in-switch replication solution that resolves read-write conflicts without coordination overhead. NetLR goes further than Harmonia by moving the entire replication functions into the network switch to realize leaderless replication without coordination overhead. Transaction Triaging (TT) [18] is a recent work that accelerates transaction processing using programmable switches. Unlike TT, NetLR does not support transactions because we target NoSQL databases like Memcached and Redis.

9 CONCLUSION

We have presented NetLR, a replicated data store architecture that achieves high performance, fault tolerance, and linearizability by leveraging the network switch as the in-network leaderless replication orchestrator. The consistency-aware read scheduling module enables us to resolve both read-write conflicts without coordination overhead. The high-performance write coordination module improves write performance for a single write by eliminating write coordinator replicas. Our active fault adaptation module enables the data store to adapt to failures quickly. We have implemented a NetLR prototype on an Intel Tofino switch and conducted extensive testbed experiments. Our experimental results have demonstrated that NetLR provides the best performance between the compared schemes including CR, Hermes, and Harmonia. In-network computing has been received great attention, and we believe that this work contributes to the database community by inspecting the potential of the switch as an in-network replication orchestrator.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for providing insightful comments. This research was partly sponsored by the National Research Foundation of Korea (NRF) grants funded by the Ministry of Science and ICT (No. 2020R1C1C1003455) and (No. 2019R1A2C2088812). Wonjun Lee is the corresponding author.

REFERENCES

[1] [n.d.]. Apache Cassandra. https://cassandra.apache.org/, Last accessed date: March 25, 2022.

[2] [n.d.]. Cavium XPliant Ethernet switch. https://www.openswitch.net/cavium/, Last accessed date: March 25, 2022.

[3] [n.d.]. A fast, compliant alternative implementation of Python. https://www.pypy.org/, Last accessed date: March 25, 2022.

[4] [n.d.]. pypacker: The fastest and simplest packet manipulation lib for Python. https://gitlab.com/mike01/pypacker, Last accessed date: March 25, 2022.

[5] [n.d.]. RocksDB: A Persistent Key-Value Store for Flash and RAM Storage. https://rocksdb.org/, Last accessed date: March 25, 2022.

[6] [n.d.]. Tofino Programmable Switch. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html, Last accessed date: March 25, 2022.

[7] 2020. Advanced Congestion & Flow Control with Programmable Switches. https://opennetworking.org/wp-content/uploads/2020/04/JK-Lee-Slide-Deck.pdf, Last accessed date: March 25, 2022.

[8] Ailidani Ailijiang, Aleksey Charapko, and Murat Demirbas. 2019. Dissecting the Performance of Strongly-Consistent Replication Protocols. In *Proc. of ACM SIGMOD*. Association for Computing Machinery, New York, NY, USA, 1696–1710.

[9] Peter A. Alsberg and John D. Day. 1976. A Principle for Resilient Sharing of Distributed Resources. In *Proc. of ICSE* (San Francisco, California, USA). IEEE Computer Society Press, Washington, DC, USA, 562–570.

[10] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 87–95.

[11] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *Proc. of USENIX FAST*. USENIX Association, Santa Clara, CA.

[12] Josiah L. Carlson. 2013. *Redis in Action*. Manning Publications Co., USA.

[13] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-Value Store. In *Proc. of ACM SOSP* (Stevenson, Washington, USA). Association for Computing Machinery, New York, NY, USA, 205–220.

[14] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. 2011. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *Proc. of ACM SIGCOMM* (Toronto, Ontario, Canada). 350–361.

[15] Ashish Gupta, Fan Yang, Jason Govig, Adam Kirsch, Kelvin Chan, Kevin Lai, Shuo Wu, Sandeep Govind Dhoot, Abhilash Rajesh Kumar, Ankur Agiwal, Sanjay Bhansali, Mingsheng Hong, Jamie Cameron, Masood Siddiqi, David Jones, Jeff Shute, Andrey Gubarev, Shivakumar Venkataraman, and Divyakant Agrawal. 2014. Mesa: Geo-Replicated, near Real-Time, Scalable Data Warehousing. *Proc. VLDB Endow.* 7, 12 (Aug. 2014), 1259–1270.

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

[17] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. In *Proc. of USENIX ATC* (Boston, MA). USENIX Association, USA, 11.

[18] Theo Jepsen, Alberto Lerner, Fernando Pedone, Robert Soulé, and Philippe Cudré-Mauroux. 2021. In-Network Support for Transaction Triaging. *Proc. VLDB Endow.* 14, 9 (may 2021), 1626–1639.

[19] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proc. of ACM SOSP* (Shanghai, China). 121–136.

[20] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. 2011. Zab: High-Performance Broadcast for Primary-Backup Systems. In *Proc. of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks (DSN '11)*. IEEE Computer Society, USA, 245–256.

[21] Antonios Katsarakis, Vasilis Gavrielatos, M.R. Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. 2020. Hermes: A Fast, Fault-Tolerant and Linearizable Replication Protocol. In *Proc. of ASPLOS* (Lausanne, Switzerland). Association for Computing Machinery, New York, NY, USA, 201–217.

[22] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133–169.

[23] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. 2016. Just Say No to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proc. of USENIX OSDI* (Savannah, GA, USA). USENIX Association, USA, 467–483.

[24] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan R. K. Ports. 2020. Pegasus: Tolerating Skewed Workloads in Distributed Storage with In-Network Coherence Directories. In *Proc. of USENIX OSDI*. USENIX Association, 387–406.

[25] Kevin Lim, David Meisner, Ali G. Saidi, Parthasarathy Ranganathan, and Thomas F. Wenisch. 2013. Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached. In *Proc. of ISCA* (Tel-Aviv, Israel). Association for Computing Machinery, New York, NY, USA, 36–47.

[26] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. 2017. IncBricks: Toward In-Network Computation with an In-Network Cache. In *Proc. of ASPLOS* (Xian, China). Association for Computing Machinery, New York, NY, USA, 795–809.

[27] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. 2008. Mencius: Building Efficient Replicated State Machines for WANs. In *Proc. of USENIX OSDI* (San Diego, California). USENIX Association, USA, 369–384.

[28] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is More Consensus in Egalitarian Parliaments. In *Proc. of ACM SOSP* (Farminton, Pennsylvania). Association for Computing Machinery, New York, NY, USA, 358–372.

[29] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *Proc. of USENIX NSDI* (Lombard, IL). USENIX Association, Berkeley, CA, USA, 385–398.

[30] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proc. of USENIX ATC* (Philadelphia, PA). USENIX Association, USA, 305–320.

[31] Jun Rao, Eugene J. Shekita, and Sandeep Tata. 2011. Using Paxos to Build a Scalable, Consistent, and Highly Available Datastore. *Proc. VLDB Endow.* 4, 4 (Jan. 2011), 243–254.

[32] Robbert Van Renesse and Fred B. Schneider. 2004. Chain Replication for Supporting High Throughput and Availability. In *Proc. of USENIX OSDI*. USENIX Association, San Francisco, CA, 91–104.

[33] Aneesh Sharma, Jerry Jiang, Praveen Bommannavar, Brian Larson, and Jimmy Lin. 2016. GraphJet: Real-Time Content Recommendations at Twitter. *Proc. VLDB Endow.* 9, 13 (Sept. 2016), 1281–1292.

[34] Jeff Terrace and Michael J. Freedman. 2009. Object Storage on CRAQ: High-Throughput Chain Replication for Read-Mostly Workloads. In *Proc. of USENIX ATC* (San Diego, California). 11.

[35] Petros Venetis, Alon Halevy, Jayant Madhavan, Marius Paşca, Warren Shen, Fei Wu, Gengxin Miao, and Chung Wu. 2011. Recovering Semantics of Tables on the Web. *Proc. VLDB Endow.* 4, 9 (June 2011), 528–538.

[36] Venkateshwaran Venkataramani, Zach Amsden, Nathan Bronson, George Cabrera III, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Jeremy Hoon, Sachin Kulkarni, Nathan Lawrence, Mark Marchukov, Dmitri Petrov, and Lovro Puzar. 2012. TAO: How Facebook Serves the Social Graph. In *Proc. of ACM SIGMOD* (Scottsdale, Arizona, USA). Association for Computing Machinery, New York, NY, USA, 791–792.

[37] Michael Whittaker, Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, Neil Giridharan, Joseph M. Hellerstein, Heidi Howard, Ion Stoica, and Adriana Szekeres. 2021. Scaling Replicated State Machines with Compartmentalization. *Proc. VLDB Endow.* 14, 11 (July 2021), 2203 –2215.

[38] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *Proc. of USENIX OSDI*. USENIX Association, 191–208.

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

[40] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan R. K. Ports, Ion Stoica, and Xin Jin. 2019. Harmonia: Near-Linear Scalability for Replicated Storage with in-Network Conflict Detection. *Proc. VLDB Endow.* 13, 3 (Nov. 2019), 376–389.