Accelerating Reads with In-Network Consistency-Aware Load Balancing

Ibrahim Kettaneh, Ahmed Alquraan, Hatem Takruri, Ali José Mashtizadeh, Samer Al-Kiswany

Abstract—We present FLAIR, a novel approach for accelerating read operations in leader-based consensus protocols. FLAIR leverages the capabilities of the new generation of programmable switches to serve reads from follower replicas without compromising consistency. The core of the new approach is a packet-processing pipeline that can track client requests and system replies, identify consistent replicas, and at line speed, forward read requests to replicas that can serve the read without sacrificing linearizability. An additional benefit of FLAIR is that it facilitates devising novel consistency-aware load balancing techniques. Following the new approach, we designed FlairKV, a key-value store atop Raft. FlairKV implements the processing pipeline using the P4 programming language. We evaluate the benefits of the proposed approach and compare it to previous approaches using a cluster with a Barefoot Tofino switch. Our evaluation indicates that, compared to state-of-the-art alternatives, the proposed approach can bring significant performance gains: up to 42% higher throughput and 35-97% lower latency for most workloads. Furthermore, our evaluation shows that our novel load balancing techniques can cope with heterogeneous load and hardware to achieve higher performance, and that FLAIR can scale to support large data sets and clusters.

Index Terms—Distributed systems, Load balancing and task assignment, Network Architecture and Design, Reliability

1 Introduction

Replication is the main reliability technique for many modern cloud services that process billions of requests each day. Unfortunately, modern strongly-consistent replication protocols [1] – such as multi-Paxos [2], Raft [3], Zab [4], and Viewstamped Replication (VR) [5] – deliver poor read performance. This is because these protocols are leader-based: a single leader replica (or leader, for short) processes every read and write request, while follower replicas (followers for short) are used for reliability only.

Optimizing read performance is clearly important; for instance, the read-to-write ratio is 380:1 in Google's F1 advertising system [6], 500:1 in Facebook's TAO [7], and 30:1 in Facebook memcached deployments [8]. Previous efforts have attempted to accelerate reads by giving read leases [9] to some [10] or all followers [11]. While holding a lease, a follower can serve read requests without consulting the leader; each lease has an expiration period. Unfortunately, this approach complicates the system's design, as it requires careful management of leases, imposes long delays when a follower holding a lease fails, and affects the write operation as all granted leases need to be revoked before an object can be modified [11].

Alternatively, many systems support a relaxed consistency model (e.g., eventual [12], [13] or read-your-write [7], [13], [14]), in exchange for the ability to read from followers, albeit the possibility of reading stale data.

In this paper, we present the fast, linearizable, network-accelerated client reads (FLAIR), *a novel protocol to serve reads from follower replicas with minimal changes to current leader-based consensus protocols without using leases, all while preserving linearizability.* In addition to improving read performance, FLAIR improves write performance by reducing the number of requests that must be handled by the leader and employing consistency-aware load-balancing.

FLAIR is positioned as a shim layer on top of a leader-based protocol (3). FLAIR assumes a few properties of the underlying consensus protocol: the operations are stored in a replicated log; at any time, there is at most one leader in the system that can commit new entries in the log; reads served by the leader are linearizable; and after committing an entry in the log, the leader knows which followers have a log consistent with its log up to that entry. These properties hold for all major leader-based protocols (Raft [3], Viewstamped Replication [5], DARE [15], Zookeeper [16], and multi-Paxos [17], [18]).

FLAIR leverages the power and flexibility of the new generation of programmable switches. The core of FLAIR is a packet-processing pipeline (Section 4) that maintains compact information about all objects stored in the system. FLAIR tracks every write request and the corresponding system reply to identify which objects are stable (i.e., not being modified) and which followers hold a consistent value for each object, then uses this information to forward reads of stable objects to consistent followers. Followers optimistically serve reads and the FLAIR switch validates read replies to detect stale values. If the switch suspects that a reply from a follower is stale, it will drop the reply and resubmit the read request to the leader.

An additional benefit of FLAIR is that it facilitates the building of novel consistency-aware load balancing techniques. In systems that grant a lease to followers [10], [11], [19] clients send read requests to a randomly selected follower. If the follower does not hold a lease, it blocks the request until it obtains a lease, or it forwards the request to the leader; either way, this approach adds additional delay. FLAIR does not incur this inefficiency as FLAIR load

• *The authors are with the David Cheriton School of Computer Science, University of Waterloo, 200 University Avenue West, Waterloo, ON, N2L 3G1. E-mails: {iskettan, ahmed.alquraan, htakruri}@uwaterloo.ca, ali@rcs.uwaterloo.ca, alkiswany@uwaterloo.ca*

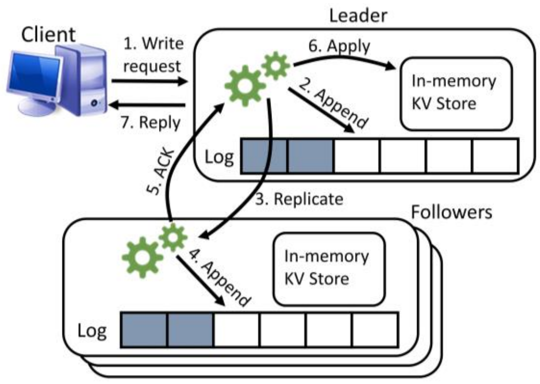


Fig. 1. The path for a write operation.

balances read requests only among followers that hold a consistent value for the requested object.

Unlike other systems that use switch new capabilities [20], [21], [22], FLAIR does not rely on the controller to update the switch information after every write operation, as this approach would add unacceptable delays. Instead, FLAIR piggybacks control messages on system replies, and the switch extracts and processes them.

In a nutshell, FLAIR implements a logically simple look-through metadata cache at a network switch. Write requests invalidate the switch metadata related to the accessed object, and write replies update the switch metadata. The switch maintains only a soft state that can be instantiated by contacting the leader replica.

Despite its simplicity, implementing this approach is complicated by the limitations of programmable switches (2) and the complexity of handling switch failures, network partitioning, and packet loss and reordering (4).

To demonstrate the powerful capabilities of the proposed approach, we prototyped FlairKV (Section 6), a key-value store built atop Raft [3]. We made only minor changes to Raft's implementation [23] to enable followers to serve reads, make the leader order write requests following the sequence numbers assigned by the switch, and expose leader's log information to the FLAIR layer. The packet-processing pipeline was implemented using the P4 programming language [24].

Our evaluation of FlairKV (7) on a cluster with a Barefoot Tofino switch shows that FLAIR can bring sizable performance gains without increasing the complexity of the leader-based protocols or the write operation overhead. Our evaluation with different read-to-write ratios and workload skewness shows that FlairKV brings up to 2.8 times higher throughput than an optimized Raft implementation, at least 4 times higher throughput compared to Viewstamped Replication, Raft, and FastPaxos, and up to 42% higher throughput and up to 35–97% lower latency for most workloads compared to state-of-the-art leases-based design [11], [19].

Compared to our previous work [25], in this paper, we evaluate three novel load balancing techniques that are built atop of FLAIR, including random, leader avoidance, and load awareness. Our evaluation shows that implementing an in-network a load balancing technique that is aware of the overhead difference between read and write operations and of heterogeneity of the cluster nodes can significantly improve performance. Furthermore, we evaluate FLAIR's scalability and show that it can scale to support large data sets and large clusters without significantly increasing the system overhead. Finally, we present a detailed proof of the safety of FLAIR.

The performance and programmability of the new generation of switches opens the door for the switches to be used beyond traditional network functionalities. We hope our experience will inform a new generation of distributed systems that co-design network protocols with systems operations.

2 Background

In this section, we present an overview of leader-based consensus protocols, followed by a look at the new programmable switches and their limitations.

2.1 Leader-based Consensus

Leader-based consensus (LC) protocols [3], [4], [5], [15], [17], [18] are widely adopted in modern systems. The idea of having a leader that can commit an operation in a single round trip dates back to the early consensus protocols [2], [26]. Having a leader reduces contention and the number of messages, which greatly improves performance [2], [17].

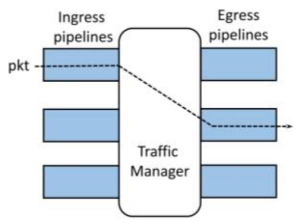
LC protocols divide time into terms (a.k.a. views or epochs). Each term has a single leader; if the leader fails, a new term starts and a new leader is elected.

Clients send write requests to the leader (1 in Fig. 1). The leader appends the request to its local log (2) and then sends the request to all follower replicas (3). A follower appends the request to its log (4) before sending an acknowledgment to the leader (5). If the leader receives an acknowledgment from a majority of its followers, the operation is considered *committed*. The leader *applies* the operation to its local state machine (e.g., in memory key-value store in Fig. 1) in (6), then acknowledges the operation to the client (7). The leader will asynchronously inform the followers that it committed the operation. Followers maintain a *commit_index*, a log index pointing to the last committed operation in the log; when a follower receives the commit notification, it advances its *commit_index* and applies the write to its local store.

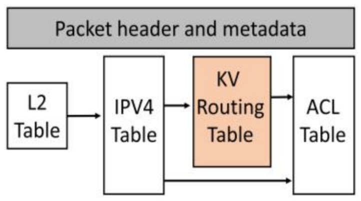
The replicated log has two properties that make it easy to reason about: it is guaranteed that if an operation at index i is committed, then every operation with an index smaller than i is committed as well; and if a follower accepts a new entry to its log, it is guaranteed that its log is identical to the leader's log up to that entry.

Client read requests are also sent to the leader. In Raft, the leader sends a heartbeat to all followers to make sure it is still the leader. If a majority of followers reply, the leader serves the read form its local store: it will check that all committed operations related to the requested object are applied to the local store before serving the request.

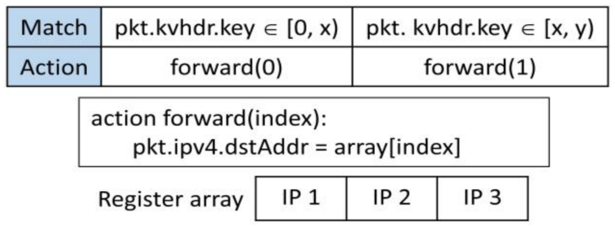
A common optimization is the leader lease optimization. Instead of collecting a majority of heartbeats for every read request, a majority of the followers can give the leader a lease [3], [17]. While holding a lease, the leader serves reads locally without contacting followers. Unfortunately,



(a) Switch data plane.



(b) Pipeline for routing based on a hash-based key



(c) Simple match-action stage for routing based on a hash-based key for the KV routing table in subfigure (b)

Fig. 2. Switch data plane.

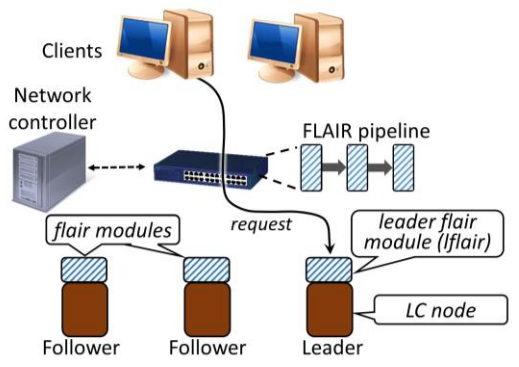


Fig. 3. System architecture. The solid arrow shows a client request, while the dashed arrow show control messages.

even with this optimization, the performance of the leader-based protocols is limited to a single-node performance.

2.2 Programmable Switches

Programmable switches allow the implementation of an application-specific packet-processing pipeline that is deployed on network devices and executed at line speed. A number of vendors produce network-programmable ASICs, including Barefoot's Tofino [27], Cavium's XPliant [28], and Broadcom Trident 3 [29].

Fig. 2.a illustrates the basic data plane architecture of modern programmable switches. The data plane contains three main components: ingress pipelines, a traffic manager, and egress pipelines. A packet is first processed by an ingress pipeline before it is forwarded by the traffic manager to the egress pipeline that will finally emit the packet.

Each pipeline is composed of multiple stages. At each stage, one or more tables match fields in the packet header or metadata; if a packet matches, the corresponding action is executed. Programmers can define custom headers and metadata as well as custom actions. Each stage has its own dedicated resources, including tables and register arrays (a memory buffer). Fig. 2.b shows a simple example of a pipeline that routes a request to a key-value store based on the key, and Fig. 2.c shows the details of the KV routing stage. The stage forwards the request based on the key in the packet's custom L4 header. The programmer implements a `forward()` action that accesses the register array holding nodes' IP addresses. An external controller can modify the register array and the table entries.

Stages can share data through the packet header and small per-packet metadata (a few hundred bytes in size) that is propagated between the stages as the packet is processed throughout the pipeline (Fig. 2.b). The processing of packets can be viewed as a graph of match-action stages.

Programmers use domain-specific languages like P4 [30] to define their own packet headers, define tables, implement custom actions, and configure the processing graphs.

Challenges. While programmable ASICs and their domain-specific languages significantly increase the flexibility of network switches, the need to execute custom actions at line speed restricts what can be done. To process packets at line speed, P4 and modern programmable ASICs have to meet strict resource and timing requirements. Consequently, modern ASICs limit (1) the number of stages per pipeline, (2) the number of tables and registers per stage, (3) the number of times any register can be accessed per packet, (4) the amount of data that can be read/written per-packet per register, (5) the size of per-packet metadata that is passed between stages. Finally, modern ASIC's lack support of loops or recursion.

3 FLAIR Overview

FLAIR is a novel protocol that targets deployments in a single data center. Fig. 3 shows the system architecture, which consists of a programmable switch, a central controller, and storage nodes. Typically, multiple FLAIR instances are deployed with each serving a disjoint set of objects. For simplicity, we present a FLAIR deployment with one replica set (i.e., one leader and its followers).

Clients. FLAIR is accessed through a client library with a simple read/write/delete interface. Read (get) and write (put) operations read or write entire objects. The library adds a special FLAIR packet header to every request, that contains an operation code (e.g., read) and a key (a hash-based object identifier).

Controller. Our design targets data centers that use a SDN network following a variant of the multi-rooted tree topology. A central controller uses OpenFlow [31] to manage the network by installing per-flow forwarding, filtering, and rewriting rules in switches.

As with previous projects that leverage SDN capabilities [20], [22], [32], [33] the controller assigns a distinct address for each replica set. The controller can use a different switch for different replica sets. The controller installs forwarding rules to guarantee that every client request for a range of keys served by a single replica set is passed through a specific switch (dubbed FLAIR switch); that switch will run the FLAIR logic for that range of keys. The controller typically selects a common ancestor switch of all replicas and installs rules to forward system replies through the same switch. Only client request/replies are routed through the FLAIR switch, leader-follower messages do not have the FLAIR header nor are necessarily routed through the FLAIR switch.

While this approach may create a longer path than traditional forwarding, the effect of this change is minimal. Li et al. [32] reported that for 88% of cases, there is no additional latency, and the 99th percentile had less than 5 μs of added latency. This minimal added latency is due to the fact that the selected switch is the common ancestor of target replicas and client packets have to traverse that switch anyway.

This approach naturally facilitates scaling the system to use multiple switches. The controller can make different switches serve different replica set, effectively load balancing the load on multiple switches.

Storage Nodes. The storage nodes run the FLAIR and LC protocols. Each node runs a FLAIR module. For read requests, before serving a read, followers verify that all committed writes to the requested object have been applied to the follower's local storage.

Write requests are processed by the leader. After a successful write operation, the leader passes to the local FLAIR module the commit index of the write and the list of followers that accepted the write operation and have a consistent log up to the commit index. The FLAIR module encodes this list into a compact bitmap and uploads it and the commit index to the switch (piggybacked on the write reply).

Programmable Switch. The switch is a core component of FLAIR: it tracks every write request and the corresponding reply to identify which objects are stable (i.e., not being modified) and which replicas have a consistent value of each object. If a read is issued while there are outstanding writes for the target object (i.e., writes without corresponding replies), the read is forwarded to the leader. If a read request is processed by the switch when there are no outstanding writes to the requested object, the switch forwards the request to one of the followers included in the last bitmap for the object sent by the leader. Followers optimistically serve read requests. The switch inspects every read reply; if it suspects that a follower returned stale data (4.4), it will conservatively drop the reply and forward the request to the leader. FLAIR forwards all writes to the leader.

FLAIR also includes techniques to handle multiple concurrent writes to the same object (4.3), packets reordering (4.7), and tolerating switch, node, and network failures (4.7).

4 System Design

FLAIR is based on the following assumptions: the network is unreliable and asynchronous, as there are no guarantees that packets will be received in a timely manner or even delivered at all, and there is no limit on the time a node or switch takes to process a packet. Clocks are not synchronized. Finally, FLAIR assumes a fail-stop failure model in which nodes and switches may stop working but will never send erroneous messages.

FLAIR assumes a few properties of the underlying consensus protocol: the operations are stored in a replicated log; at any time, there is at most one leader in the system that can commit new entries in the log; and after committing an entry in the log, the leader knows which followers have a log consistent with its log up to that entry. If an operation at index i in the log is committed, then every operation with an index smaller than i is committed as well. If a follower accepts a new entry to its log, then it is guaranteed that the follower log is identical to the leader's log up to that entry.

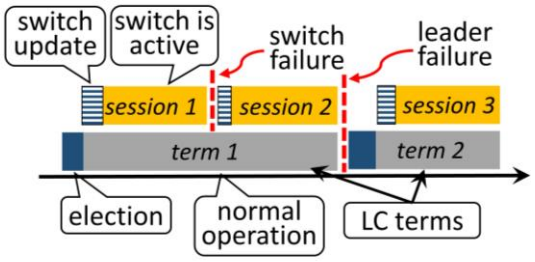


Fig. 4. FLAIR sessions. Time is divided into terms. Each term starts with a leader election. Each term has one or more sessions that start with updating the switch data.

We note that all major leader-based consensus protocols (e.g., Raft [3], Viewstamped Replication [5], [34], DARE [15], Zab [16], and multi-Paxos implementations [17], [35]) hold these properties.

The underlying consensus protocol divides time into terms. Each term has a single leader; if the leader fails, a new term starts and a new leader is elected. FLAIR further divides time into sessions (Fig. 4). During a session the leader is bonded to a single switch that runs the FLAIR pipeline. A session ends when a leader fails or the leader suspects that the switch has failed. An LC term may have one or more sessions, but a session does not span multiple terms.

4.1 Network Protocol

Packet format. FLAIR introduces an application-layer protocol embedded in the L4 payload of packets. FLAIR uses UDP to issue client requests in order to achieve low latency and simplify request routing. Communication between replicas uses TCP for its reliability. A special UDP port is reserved to distinguish FLAIR packets; for UDP packets with this port, the switch invokes the FLAIR custom processing pipeline. Other switches do not need to understand the FLAIR header and will treat FLAIR packets as normal packets. In this way, FLAIR can coexist with other network protocols.

Fig. 5 shows the main fields in the FLAIR header. We briefly discuss the fields here (a detailed discussion of the protocol is presented next):

- OP: the request type. Clients populate this field in the request packet (e.g., read, or write); replicas populate this field in the reply packets (e.g., read_reply, write_reply).
- KEY: hash-based object identifier.
- SEQ: a sequence number added by the switch. The switch increments the sequence number on every write request.
- SID: a unique session id. The <SID, SEQ> combination represents a unique identifier for every write request.
- LOG_IDX: a log index. In a write_reply, the log index is the index at which the write was committed. For reads, the switch populates LOG_IDX to make sure the followers' logs are committed and applied up to that index.

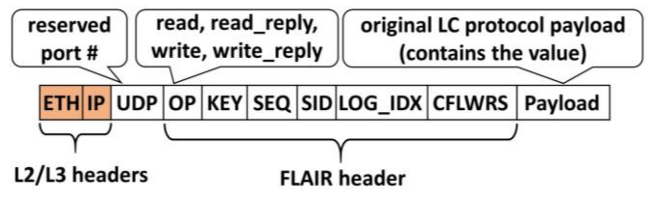

reserved port #
read, read_reply, write, write_reply
original LC protocol payload (contains the value)
ETH
IP
UDP
OP
KEY
SEQ
SID
LOG_IDX
CFLWRS
Payload
L2/L3 headers
FLAIR header

Fig. 5. FLAIR packet format.

```
SessionArrayEntry {                 KGroupArrayEntry {
  bit<1>   is_active;                 bit<1>   is_stable;
  bit<32> session_id;                 bit<64> seq_num;
  bit<32> leader_ip;                  bit<64> log_idx;
  bit<64> session_seq_num;            bit<8>   consistent_followers;
  bit<48> heartbeat_tstamp;         }
}
```

Listing 1. Session and kgroup entries. The numbers indicate the field size in bits.

- CFLWRS: In write_reply, the CFLWRS is a map of the followers that have a consistent log up to LOG_IDX.

Following the FLAIR header is the original LC protocol payload, which includes the value for read/write request.

4.2 Switch Data Structures

To process a read request, the switch performs two specific tasks (4.4). First, it forwards read requests to consistent followers while balancing the load among them. Second, it verifies the read replies to preserve safety. To perform these tasks, the switch maintains two data structures: a session array and a key group array.

Session array. A single switch typically supports multiple replica sets (i.e., FLAIR+LC instances) with each set storing a disjoint set of keys. Each entry in the session array maintains the session status for a single replica set. An entry contains an is_active flag, session id, leader IP address, current session sequence number, and the timestamp of the last heartbeat received from the leader FLAIR module (dubbed *lflair*) (Listing 1). When is_active is true, we say the session is *active*, which indicates that the session entry and kgroup array are consistent with the leader's information. The switch processes packets using the FLAIR custom pipeline only if the session is active; otherwise, it will drop all FLAIR packets, rendering the system unavailable to clients until the switch can reach the *lflair* module and sync its session entry and key group array.

Key group (KGroup) array. To decide if followers can serve a certain read request, the switch needs to maintain information about which followers have the latest committed value of every object. Maintaining such information in the switch ASIC's memory is not feasible; instead, FLAIR groups objects based on their key and maintains aggregate information per group. We use the most significant k bits of the key to map an object to a key group (kgroup).

Every FLAIR+LC instance has a dedicated kgroup array. Each entry in the array (Listing 1) contains the status of a single kgroup, including an is_stable flag that indicates if all objects in the kgroup are stable. If a kgroup is not stable (is_stable is false), this indicates that at least one object in the kgroup is being modified (i.e., has an outstanding write in the system). The array entry also includes the sequence number (seq_num) of the last write request processed by the switch for any object in the kgroup, the log index (log_idx) of the last successful write to any object in the kgroup, and the consistent_followers bitmap pointing to all followers that have a consistent log up to log_idx.

4.3 Handling Write Requests

To issue a write request, a client populates the OP and KEY fields of the FLAIR packet header and puts the value in the payload, then sends the request.

When the switch receives the request, it will mark the corresponding kgroup entry as unstable. The switch will increment the session_seq_num in the session array and use it to populate the sequence number (seq_num) in the kgroup entry and the sequence number (SEQ) in the request header. Finally, the switch populates the session id (SID) field in the header and forwards the request to the leader.

The *lflair* module will verify that the session id is valid and will pass the write request to the leader. The leader verifies that the <SID, SEQ> combination is larger than the <SID, SEQ> number of any previous write request it ever received, else it will drop the packet. The LC leader will process the write request following the LC protocol (2.1): it will replicate the request to all followers, and when a majority of followers acknowledge the operation, the write operation is considered committed. A follower will acknowledge a write operation only if its log is identical to the leader's log up to that entry.

For the write reply, the leader will pass the following to the *lflair* module: the LC protocol payload for the write_reply, the log index at which the write was committed, and the list of followers that acknowledged the write. The *lflair* module will create the write reply packet with the leader provided payload, and will populate the LOG_IDX and the bitmap of the consistent followers (CFLWRS) using the information provided by the leader. *lflair* module populates the sequence number (SEQ) in the write_reply header using the SEQ of the corresponding write request. The *lflair* module then sends the write_reply packet.

The switch will process the write_reply header and verify its session id. The switch will compare the sequence number (SEQ) of the reply to the sequence number (seq_num) in the kgroup entry; if they are equal, this signifies that no other write is concurrently being processed in the system for any object in the kgroup. Consequently, it will update the log_idx and the consistent_followers fields in the kgroup entry using the values in the write reply. Then it will mark the kgroup stable and forward the reply to the client.

If the sequence number in the reply is smaller than the sequence number in the kgroup entry, this indicates that a later write to an object in the same kgroup has been processed by the switch. In this case, the switch forwards the write reply to the client without modifying the kgroup entry. The kgroup entry remains unstable until the last write (with a SEQ number in the write_reply equal to the seq_num in the kgroup entry) is acknowledged by the leader.

In a nutshell, the switch acts as a look-through metadata cache. Write requests invalidate the switch metadata related to the accessed kgroup, and write replies update the kgroup metadata at the switch. An additional advantage of this approach is that by hosting a simple cache at the switch we can consistently load balance reads across followers without substantially effecting complexity.

4.4 Handling Read Requests

Clients fill the OP and KEY fields of the FLAIR header and send the request. When the switch receives the request, it will check the kgroup entry. If the entry is stable, the switch will fill the sequence number (SEQ) and log index (LOG_IDX) header fields using the values in the kgroup entry. Then it will forward the request to one of the followers indicated in the consistent_followers bitmap. 6.2 details our load balancing techniques.

If the kgroup entry is not stable, the switch forwards the read request to the leader. We note that there is a chance for false positives in this design, as a single write will render all the objects in the same kgroup unstable. This is a drawback of maintaining information per group of keys. This inefficiency is incurred by leases-based protocols as well, as they maintain a lease per group of objects.

When a follower receives a read request, the follower's FLAIR module validates the request, then calls advance_then_read(LOG_IDX, key) routine, which compares the follower's commit_index to LOG_IDX. If the commit_index is smaller, the follower advances its commit_index to equal LOG_IDX, apply all the log entries to the local store, then serve the read request. The FLAIR module will populate the read_reply header; for the SEQ and SID fields, it will use the values found in the read request header.

We note that it is safe to advance the follower's commit_index to match the LOG_IDX in the read request, as the switch forwards read requests to a follower only if the leader indicates that all entries in the log up to that log index are committed, and that this specific follower is one of the replicas that have a log consistent to the leader's log up to that index. We discuss FLAIR correctness in 5 .

When the switch receives a read_reply from a follower, it validates the session id, then verifies that the SEQ number of the read_reply equals the seq_num of the kgroup entry. If the sequence numbers are not equal, this signifies that a later write request was processed by the switch and there is a chance the follower has returned stale value. In this case, the switch drops the read_reply, generates a new read request using the KEY field from read_reply packet, and submits the read request to the leader. If the sequence number of the reply equals the sequence number in the kgroup entry, the switch forwards the reply to the client.

If a read request is forwarded to the leader, the *lflair* module verifies the session id, then calls advance_then_read (LOG_IDX, key). The switch verifies that the leader reply is valid (i.e., has the correct session id) before forwarding it to the client.

4.5 Load Balancing

FLAIR facilitates designing load balancing policies that are data consistency aware. We designed three such load balancing techniques that choose which replica from the list of consistent replicas will serve a read request.

- *Random*. This technique selects a replica to serve a read request in random fashion from the list of consistent followers.
- *Leader avoidance*. Our benchmarking revealed that the write operation takes 35 times longer than a read operation; most of this overhead is borne by the leader. Consequently, this load-balancing technique avoids sending read requests to the leader for stable kgroups if there are any writes in the system. The aim is to reduce the leader load, as it is already busy serving writes and serving reads for unstable kgroups.
  We can detect if a leader is serving any writes by comparing the sequence number of a write_reply with the session_seq_num. If they are not equal, then there are pending writes in the system and the leader should not be burdened with any reads to stable kgroups.
- *Follower load awareness*. This technique distributes the load across followers proportionally to their load in the last n seconds. This technique is especially useful for deployments that use heterogeneous hardware, experience workload variations, or deploy more than one replica (i.e., for different key ranges) on the same machine.

4.6 Session Start Process

On the start of a new session, the *lflair* module reads the last session id from the LC log, increments it, and commits the new session id to the LC log. Then the *lflair* module asks the central controller for a new switch. The central controller neutralizes the old switch (making it drop all FLAIR packets) and reroutes FLAIR packets to a new switch, then confirms the switch change to the *lflair* module. This step guarantees that at any time at most one FLAIR switch is active. The *lflair* module updates the session entry (Listing 1) at the switch with the current leader IP and session id. For each new session, session_seq_num is reset to zero.

Populating the kgroup array. The *lflair* module maintains a copy of the kgroup array similar to the one maintained by the switch. If the leader did not change between sessions (e.g., the session change is due to switch failure), the kgroup array at the *lflair* module is up-to-date. The *lflair* module will set the seq_num entry in all kgroup entries to zero (equal to the session_seq_num in the session entry), and upload it to the switch.

If the kgroup array at the *lflair* module is empty – for instance, after electing a new leader – the *lflair* module will query the leader for three pieces of information: its commit_index, the list of followers with the same commit_index, and a list of all uncommitted operations in the log (i.e., the operations after the commit_index in the log). The list of uncommitted operations is typically small, as it only includes operations that were received before the end of the last term but were not committed yet. The *lflair* module will traverse the list of uncommitted writes and mark their target kgroup entries unstable. For all other kgroup entries, the *lflair* module will mark them stable and set their seq_num to zero, log_idx to the leader's commit_index, and consistent_followers to include all the followers that have the same commit_index as the leader's. After updating the session entry and the kgroup array at the switch, the *lflair* module activates the switch session (sets is_active to true).

4.7 Fault Tolerance

Follower Failure. We rely on the LC protocol to handle follower failures. To avoid sending read requests to a failing follower, the leader notifies the *lflair* module when it detects the failure of a follower. The *lflair* module removes the follower from the switch-forwarding table (3).

Leader Failure. On leader failure, a new leader is elected and a new term starts. The new leader informs the *lflair* module of the term change; and the *lflair* module starts a new session (3).

The *lflair* module sends periodic heartbeats to the switch. Upon receiving a heartbeat, the switch determines whether it is from the current session. If the heartbeat is valid, the switch updates the heartbeat_timestamp in the session array and replies to the *lflair* module.

Switch Failure. If the *lflair* module misses three heartbeats from the switch, the *lflair* module will suspect that the switch has failed and will start a new session (4.5). For efficiency (i.e., does not affect safety), if the switch misses three heartbeats from the leader, it will deactivate the session.

Network Partitioning. If a network partition isolates the switch from the leader, the leader treats it as a failed switch, as detailed above. If a network partition isolates the switch from a follower, read requests forwarded to the follower will time out and the client will resubmit the request. This failure affects performance, but not correctness. Upon determining that a follower is not reachable, the leader removes it from the forwarding table, as in the case of the failed follower described above.

Packet Loss. If a read or write request is lost, the client times out and resubmits the request. If a write reply is lost before reaching the switch, the kgroup entry will remain unstable until a new write operation to any key in the kgroup succeeds. While the kgroup entry is not stable, all read requests are forwarded to the leader.

Packet Reordering. It is critical for FLAIR's correctness that the leader processes write requests in the same order that they are processed by the switch. Every write operation gets a unique <SID, SEQ> number. The switch marks a kgroup entry unstable until the leader replies to the last write issued for a key in the kgroup. Consequently, if the leader processes the requests out of order, the switch will incorrectly mark a kgroup stable while the out-of-order writes modify its objects. To prevent this scenario, the leader keeps track of the largest <SID, SEQ> it has ever processed and drops any write request with a smaller number. While session numbers (SIDs) are maintained in the log, the largest processed sequence number is retained in memory. If the leader fails, the new leader starts a new session, increments the session id (SID), and sets the session sequence number (SEQ) to zero.

5 Correctness

FLAIR only adds the ability to serve reads from followers. In this section we present an informal discussion of the safety of the read operations in FLAIR. Furthermore, we used the TLA+ model checking tool to verify the FLAIR correctness. The TLA specification is available in our technical report [36].

FLAIR processes read and write requests only when the switch is in an active state. We say the switch is *active* if it has an active leader-switch session, meaning the leader and the switch did not miss three consecutive heartbeats from each other. This signifies that the switch information is up-to-date with the *lflair* module's information. We first discuss safety during the active state then we discuss the safety during failure scenarios.

5.1 Safety during an Active Session

The correctness condition for reads is that the value returned by FLAIR is identical as if the read was served by the leader. This is guaranteed using the following two steps.

First, the switch only forwards read requests to followers when the kgroup entry is stable. The switch assigns a unique and strictly increasing sequence number for every write request. The switch keeps track of the sequence number of the last write operation in the *wlseq* field in the kgroup entry. The leader processes writes in an increasing order of sequence numbers. The leader ignores write requests with a sequence number smaller than the sequence number of the last write it received. The leader includes the sequence number of the write request in the write reply.

We say a kgroup is *stable* if the switch receives a write reply with a sequence number equal to *wlseq*. This signifies that there are no on-the-fly writes in the system that can change an object's value since the leader processes requests in the order of sequence numbers. Hence, the last leader-provided consistent_followers bitmap points to followers that have the last committed value for every object in the kgroup. The kgroup stays in the stable state until the switch receives a write request for an object in the kgroup, then the kgroup becomes unstable.

If a kgroup is stable, FLAIR may forward read requests to one of the replicas included in the last leader-provided consistent_followers bitmap. Since this is the last list provided by the leader and there are no later writes in the system, all followers in the list are consistent with the leader for this kgroup and will serve values identical to the value at the leader.

Second, after forwarding a read request to a follower (say, follower A), the switch may receive a write request that modifies the object. The leader may replicate the write request to a majority of nodes that does not include A. If the leader processes the write request before A serves the read request, A will return stale data. To avoid this case, followers include in the read reply the last sequence number that modified a kgroup. The switch performs a safety check on every read reply coming from followers: it verifies that the kgroup is still stable, and that the sequence number in the read_reply is equal to the sequence number in the kgroup entry. If the sequence numbers do not match (which indicates that there are later writes to objects in the kgroup), the switch resends the read request to the leader.

5.2 Safety under Failure Scenarios

Leader Failure. If a leader fails, the switch misses three heartbeats from the leader and changes to an inactive state. The switch drops all FLAIR requests during the inactive

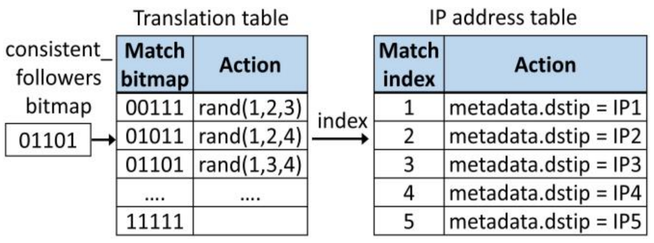


Fig. 6. Logical view of the forwarding logic. The stability bitmap matches an entry in the translation table and executes the corresponding action, generating an index of the selected destination's IP address. Using the index, the IP address table sets the destination's IP address in the metadata.

state rendering the system unavailable. When a new leader is elected, it will send heartbeats with a new leader_ip and session_id. The switch will detect that this is a new leader and will start the session startup process before entering an active state.

Switch Failure. If a switch fails, a new switch will be selected to run the FLAIR pipeline (4.6). The switch starts in an inactive state. The session start process syncs the switch with the leader before switching to active state.

Packet Loss. Two packets modify the switch metadata: write requests and replies. If a write request is dropped after it was processed by the switch, a client will eventually time out and repeat the request. The kgroup stays unstable until a new write is received and processed. Consequently, this only impacts performance but not safety. If a write reply packet is dropped before reaching the switch, the kgroup will stay unstable until a future write is processed. Again, this only affects performance, not safety.

Follower Failure. If a follower fails while processing a read request, the client will time out and repeat the read request. The resubmitted request is handled as a new request. All other follower failure scenarios are handled by the underlying consensus protocols.

6 Implementation

To demonstrate the benefits of the new approach, we prototyped FlairKV, a FLAIR-based key-value store built atop Raft [23]. We chose Raft due to its adoption in production systems and the availability of standalone production-quality implementations [37].

6.1 Storage System Implementation

We have implemented FlairKV, including all switch data plane features, the FLAIR module, leaders' and followers' modifications, and the client library. We extended the Raft's follower code to implement an advance_then_read() function. We extended the leader to notify the lflair module as soon as it gets elected, and to extract its commit_index, the list of followers with a commit_index equal to the leader's commit_index, and the list of uncommitted writes. We extended the write reply with the list of followers which acknowledged the write. We implemented the leader lease optimization [3], [17] and modified Raft's client library to add the FLAIR header to client requests.

6.2 Switch Data Plane Implementation

The switch data plane is written in P4 v14 [24] and is compiled for Barefoot's Tofino ASIC [27], with Barefoot's P4Studio software suite [38]. Our P4 code defines 30 tables and 12 registers: six for the session array and six for the kgroup array. The kgroup array has 4K entries. Larger number of kgroups had negligible effect on performance. In total, our implementation uses less than 5% of the on-chip memory available in the Tofino ASIC, leaving ample resources to support other switch functionalities or more FlairKV instances. The rest of this section discusses optimizations implemented in FlairKV to cope with the strict timing and memory constraints of P4 and switch ASIC.

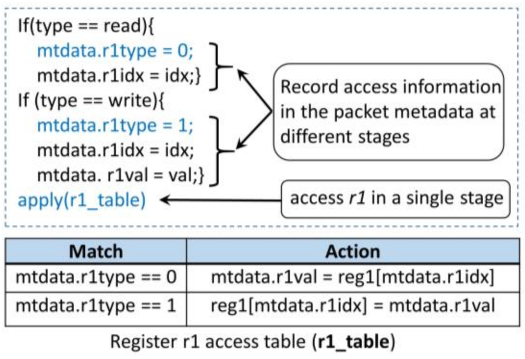
Heartbeats implementation. The leader and the switch exchange periodic heartbeats. If the switch misses three heartbeats from the leader, the switch deactivates the session. Instead of running a process in the controller to continuously track heartbeats, the switch monitors missed heartbeats as part of the validation step in the processing pipeline. The switch keeps track of the timestamp of the last heartbeat received in the session array (Listing 1). When processing any FLAIR packet, the switch computes the difference between the current time and the last heartbeat timestamp; if the difference is larger than three heartbeats, the switch deactivates the session, making the system unavailable until the leader starts a new session.

Forwarding logic translates the consistent followers' bitmap to follower IP addresses. Storing the IP addresses of consistent followers for every entry in the kgroup array significantly increases the memory footprint. Moreover, randomly selecting a follower from the list while avoiding inconsistent ones is tricky given the P4 and current ASIC challenges (2.2). Instead, the FlairKV leader encodes the follower status in a one-byte consistent_followers bitmap (Listing 1). Replicas are ordered in a list. If the least significant bit in the consistent_follower bitmap is set, this indicates that the first replica in the list is consistent, and so forth.

When forwarding a read request, the switch translates the encoded bitmap of consistent followers to select one follower; Fig. 6 shows the translation process. The consistent_followers bitmap is used as an index to the translation table. Each entry in the table has an action that randomly selects a number that is then used as an index to the IP addresses table.

This design has two benefits: it significantly reduces the memory footprint of the kgroup array, and it can be accelerated using P4 "action profiles" [39].

Load balancing. In our implementation of the follower load awareness load balancing technique followers report the length of the request queue in every heartbeat. Every second, the leader calculates the average queue length for each follower and assigns proportional weights to each follower. The leader updates the translation table (Fig. 6) to reflect these weights. For instance, if follower 1 should receive double the load of any other replica, the action for a bitmap 00111 will be rand(1, 1, 2, 3), doubling the chance replica 1 is selected.



```
If(type == read){
  mtdata.r1type = 0;
  mtdata.r1idx = idx;}
If (type == write){
  mtdata.r1type = 1;
  mtdata.r1idx = idx;
  mtdata. r1val = val;}
apply(r1_table)
```

Record access information in the packet metadata at different stages
access *r1* in a single stage

Match	Action
mtdata.r1type == 0	mtdata.r1val = reg1[mtdata.r1idx]
mtdata.r1type == 1	reg1[mtdata.r1idx] = mtdata.r1val

Register r1 access table (**r1_table**)

Fig. 7. **Register access table.** P4 code aggregates access information that is used by a dedicated register access table.

Register access logic. Each stage has its own dedicated registers, and a register can be accessed only once in a stage. This restriction complicates FlairKV's logic, as different packet types (e.g., read and write_reply) must access the same registers at different stages in the pipeline. To cope with this restriction, FlairKV adds a dedicated table to access each register. Fig. 7 shows an example of an action table for accessing register *r1*. Our code aggregates the information about all possible modes of accessing *r1* in the packet's metadata, including the access type (read or write), the index, and which data should be written or where the value should be read to. We then use a dedicated match-action table (Fig. 7) to perform the actual read or write operation to/from the register in a single stage with a single invocation of the table. This approach has the additional benefit of reducing the number of stages.

Processing concurrent requests. The switch processes packets sequentially in a pipeline. Each pipeline stage processes one packet at a time. The switch may have multiple pipelines, each serving a subset of switch ports. FLAIR uses a single ingress pipeline and all egress pipelines. If a FLAIR packet is received on a different ingress pipeline, the packet is recirculated [39] to the FLAIR pipeline.

7 Evaluation

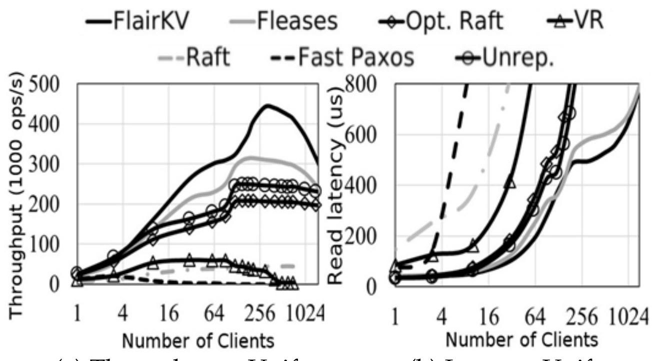
We compare our prototype with previous approaches in terms of throughput and latency (7.1) with different workload skewness and read/write ratios (7.2). Then, we evaluate FLAIR's performance under different failure scenarios (7.3), scalability (7.4), load-balancing performance (7.5), and performance with larger data sets (7.6).

Testbed. We conducted our experiments using a 13-node cluster. Each node has an Intel Xeon Silver 10-core CPU, 48 GB of RAM, and 100 Gbps Mellanox NIC. The nodes are connected to an Edgecore Wedge 100 ×32BF switch with 32 100 Gbps ports. The switch has Barefoot's Tofino ASIC, which is P4 programmable. Unless otherwise specified, three machines ran the server code, while the other 10 machines generated the workload.

Alternatives. We compare the throughput and latency of the following designs and optimizations:

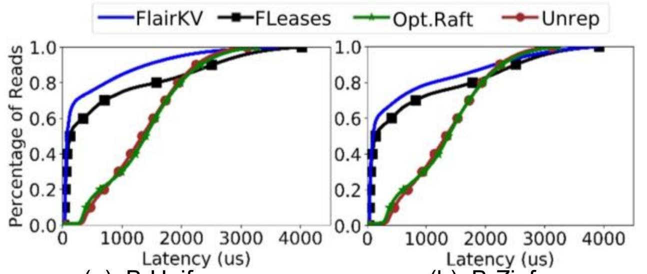
- Leader-based. We used two leader-based protocol implementations: LogCabin [23] and Viewstamped Replication (VR) [40].
- Optimized Leader-based (Opt. Raft). Our benchmarking revealed that the original Raft implementation could not utilize the resources of our cluster. We implemented two main optimizations: first, we changed the request-processing logic from an event-driven to a thread-pool design, as our benchmarking indicated a thread-pool performs better; second, we implemented the leader-lease optimization. These changes significantly improved Raft's performance.
- Fast Paxos. An alternative to the leader-based design is the quorum design. Client read requests are sent to all followers, and each follower responds directly to the client. The client waits for a reply from a supermajority [41] before completing a read. We used a Fast Paxos implementation that implements only the normal case [40].
- Follower-lease optimization (FLeases). Similar to MegaStore [11], the leader grants read leases to all followers. Before serving a write, the leader revokes all leases, processes the write operation, and then grants a new lease to followers. The lease's grant/revoke messages are piggybacked on the consensus protocol messages. However, writes should be processed by all followers before replying to the client. In our experiments, if a follower receives a read request for an object for which it does not have an active lease, it forwards the request to the leader. MegaStore applications typically partition the keys into groups, each group contains logically-related keys [11] (e.g., a key group per blog [11]). We partitioned the keys into 4K groups (the same number of kgroups in FlairKV), and followers get a lease per group. Clients randomly select a follower for each read request and send the request directly to it.
- Unreplicated/NOPaxos (Unrep.). We use an unreplicated Optimized-Raft on a single node as a baseline. The single node stores the data set and serves all operations without replication. This configuration also represents the best possible performance of the network-optimized NOPaxos [32] protocol. NOPaxos uses a network switch to order and multicast operations to all replicas. An operation is successful if the majority accepts a write or returns the same value for a read. Consequently, NOPaxos read performance is limited by the slowest node in the majority of nodes. NOPaxos evaluation shows that the best throughput and latency the protocol can achieve are within 4% of an unreplicated system [32].
- FlairKV. Unless otherwise specified, we used FlairKV with the leader-avoidance load-balancing technique.

We benchmarked every system and selected a configuration that maximized its performance. We stored all data in memory. In all experiments, all systems' performance (with the exception of FastPaxos) was stable with a standard deviation less than 1%.



(a) Throughput - Uniform (b) Latency-Uniform

Fig. 8. Throughput and Latency while varying the number of clients for workload B for the uniform distribution



(a) B-Uniform (b) B-Zipf

Fig. 9. Latency CDF. The **figures** show the latency CDF for reads under workload B using 300 clients with a uniform distribution (a), and a Zipf distribution with skewness of 0.99 (b). The lines for Opt. Raft and Unrep. almost overlap.

Workload. We used synthetic benchmarks and the YCSB benchmark [42] to evaluate the performance of all systems. In our evaluation, we considered both uniform and skewed workloads. The skewed workload follows the Zipf distribution with a skewness parameter of 0.99. We present our results with a data set of 100,000 keys. We present our results with two additional data sets, 1 million and 4 million keys, in 7.6. The key size is 24 bytes, and the hash of the key string is used as the key in the FLAIR protocol. The value size is 1 KB.

7.1 Performance Evaluation

We compared the seven systems using YCSB workload B (95:5 read:write ratio) while varying the number of clients, with uniform and skewed workload distribution. Fig. 8 shows the throughput and average latency with a uniform distribution. FlairKV achieves up to 42% higher throughput and 23.7% lower average latency than FLeases, and 1.3 to 2.1 times higher throughput and 1.5 to 2.4 times lower latency compared to optimized Raft and unreplicated setup. Fast Paxos, Raft, and VR, achieve the lowest throughput and highest latency as these systems contact the majority of nodes for every read. FlairKV performance has a similar pattern under skewed workloads [25].

FlairKV achieves better performance than FLeases for three reasons. First, FlairKV uses the leader-avoidance load-balancing technique, which reduces the load on the leader when there are writes, thereby accelerating writes and shortening the time period in which kgroups are marked unstable. This approach is effective as writes take almost 35 times longer than reads in Opt.Raft, and 30 times longer in the unreplicated setup. We recorded the number of read requests served by the leader. For instance, with 300 clients (Fig. 8.a) the leader served 2% of the reads in FlairKV (those are reads to unstable kgroups), while it served 34% of the reads in FLeases. We note that the leader-avoidance technique cannot be applied to FLeases which tasks the clients with selecting a follower to send the read request to. This technique requires accurate information about the current load of the leader and which followers are stable which are not available to clients.

Second, in FLeases, when an object is not stable, if a client sends a request to a follower, the follower will redirect the request to the leader, increasing overhead and incurring extra latency. Unlike FLeases, FlairKV switch knows if an object is not stable and forwards read requests for that object directly to the leader. The third reason which had a minor impact when using three replicas is that the write operation in FLeases needs to reach all followers, while FlairKV writes only need a majority.

Optimized-Raft's performance is better than that of Raft, VR, and FastPaxos. The unreplicated deployment slightly improves throughput and latency over Optimized-Raft by avoiding the replication overhead for write operations. These two systems still lag behind FlairKV as they only utilize a single node (the leader) for serving all reads and writes.

We note that all systems have a dip in the throughput curve at high number of clients. This is a side effect of using a thread-per-request server design which has a high overhead with large number of clients. Using a thread-pool design should eliminate this performance dip.

Latency evaluation. Fig. 9.a shows the latency CDF of FlairKV, FLeases, OptRaft, and Raft. Under the uniform workload B with 300 clients (other workloads had similar results). FlairKV lowered the latency for the slowest 40% requests by at least 38% relative to FLeases. Under the Zipfian workload (Fig. 9.b), FlairKV lowered the slowest 50% of request by up to 35% relative to FLeases.

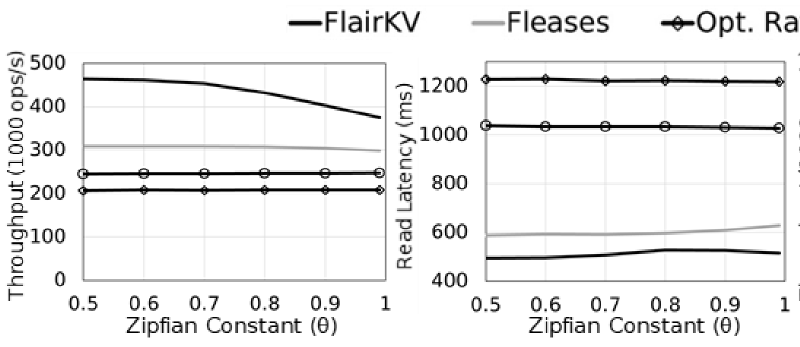
FLeases has higher latency as it incurs extra delay due to the load imbalance between nodes (e.g., the leader serves 41% of requests for workload B with Zipf distribution) and due to followers redirecting 4% of requests to the leader.

Under all workloads, FlairKV significantly improved operation's latency relative to Opt.Raft and Raft. The median latency of FlairKV is 2% of Raft's latency and 2-8% of OptRaft's latency.

7.2 Workload Variations

We measured the impact of two workload variations: skewness (Fig. 10) and read/write ratios (Fig. 11). We vary the Zipfian constant from 0.5 to 0.99. FlairKV consistently achieves better performance: 1.26 to 2.25 times higher throughput and 1.13 to 2.48 times lower average latency compared to all other systems.

Our evaluation with different read to write ratios (Fig. 11) shows that FlairKV has up to 1.5 times higher throughput for all read to write ratios, with the exception of the read-only workload in which their performance is compa-



(a) Throughput
(b) Read latency

Fig. 10. Throughput and Latency while varying skewness. The figures show the throughput (a) and the average latency (b) for different zifpian constants for a uniform workload B with 300 clients.

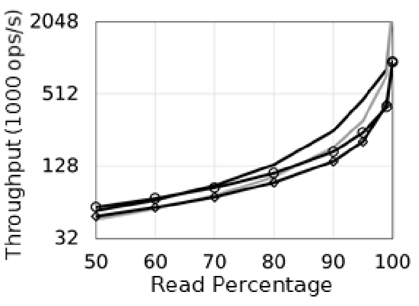


Fig. 11. Throughput while varying read ratio. Using uniform workload B

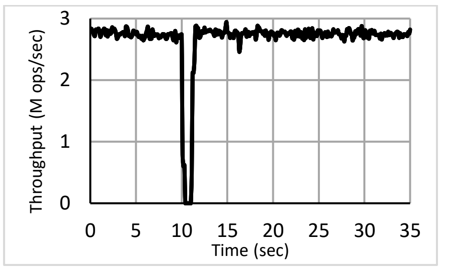


Fig. 12. Throughput during a switch failover.

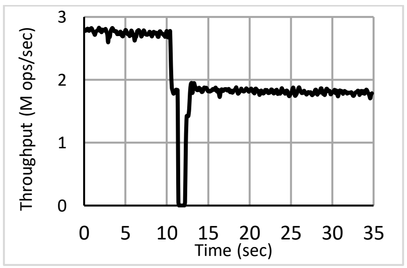


Fig. 13. Throughput during leader failover.

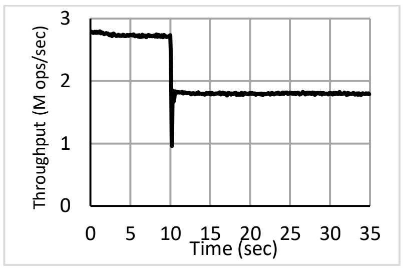


Fig. 14. Throughput during a follower failure.

rable. FlairKV has 1.25 to 2.8 times higher throughput compared to the Opt. Raft. Compared to the unreplicated setup, FlairKV has up to 2.8 times higher throughput for workloads with 70% reads or more and a comparable performance under write heavy workloads (read ratio 50-70%).

7.3 Fault Tolerance

To demonstrate FlairKV fault tolerance techniques, we measured the system throughput using workload C under three failure scenarios: switch, leader, and follower failure.
Switch Failure. We ran FlairKV at peak throughput for 35 seconds (Fig. 12). At the 10s mark, the controller emulated a switch failure by wiping out the switch registers and installing rules to drop switch heartbeats. After missing 3 heartbeats, the leader suspects that the switch has failed and starts a new session. During this process, the switch is inactive, which causes the throughput to drop to zero for 750ms. Afterwards, the switch resumes normal operations.
Leader Failure. Fig. 13 shows FlairKV throughput during the leader failure. We ran FlairKV at peak throughput for 35 seconds. At the 10s mark, we kill the leader process. Write requests fail, but the switch continues to forward read requests to followers. After missing 3 heartbeats the switch deactivates the session, and the throughput drops to zero. After 6 heartbeats, the followers elect a new leader that starts a new session. The system resumes its operation with one leader and one follower.
Follower Failure. We ran FlairKV at peak throughput for 35 seconds (Fig. 14). At the 10s mark, we kill a follower process. This causes a drop in throughput as fewer replicas are available to serve read requests. The switch keeps forwarding client requests to the failed follower until the leader updates the switch. The dip in throughput at the second 10 is because we use closed-loop clients and some of the clients block waiting for the failed replica before timing out and retrying. Afterwards, the system throughput drops by 33% due to the loss of one follower.

7.4 Scalability

To demonstrate FlairKV scalability, we measured the system throughput using a read-only YSCB workload C while varying the number of replicas (Fig. 15). The figure shows that FlairKV throughput scales linearly with the number of replicas, reaching 5.4 million request per second with 6 followers. We notice that the system achieves much higher performance under the read-only workload mainly due to the lower operation overhead (as writes take 35 times longer than reads even without accounting for the replication overhead). FlairKV is almost perfectly scalable, it only deviates by 1.1% from perfect linearly scalable performance.

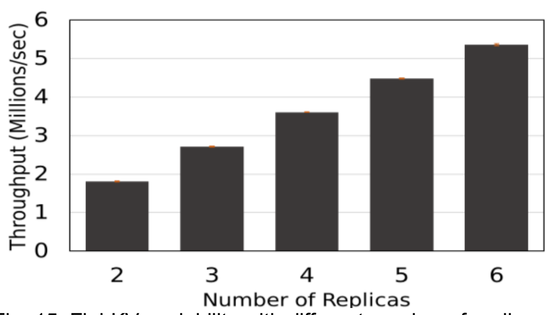


Fig. 15. FlairKV scalability with different number of replicas

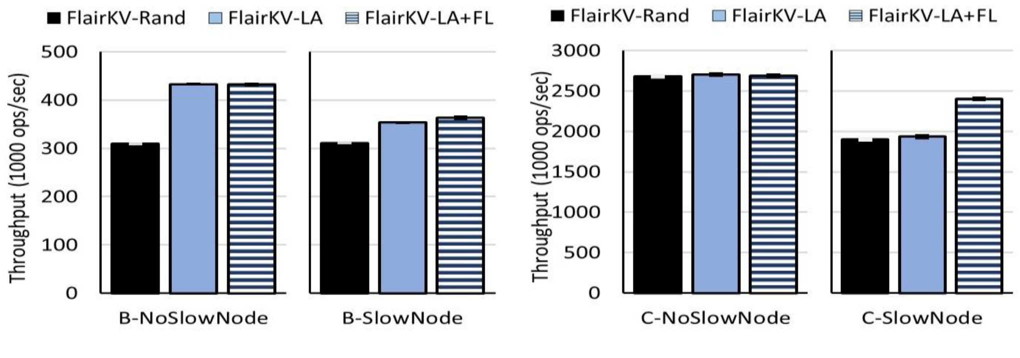


Fig. 16. Throughput using different load-balancing techniques. (a) Uses workload B without slowing any follower. (b) Uses workload B and slows one follower. (c) uses workload C without slowing any follower. (d) Uses workload C and slows one of the followers.

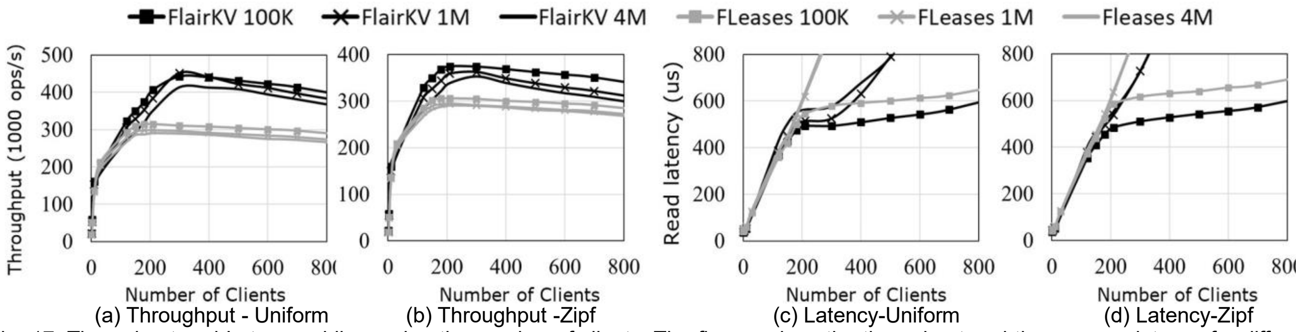


Fig. 17. Throughput and Latency while varying the number of clients. The figures show the throughput and the average latency for different number of keys for workload B for the uniform distribution (a, c), and for the Zipf distribution (b, d).

7.5 Load-balancing Performance Evaluation

We measured the system throughput using the following three configurations of FlairKV (detailed in 6.2)

- *FlairKV*-Rand selects a follower or the leader at random. Consequently, read requests for stable kgroups are uniformly spread across the followers and the leader.
- *FlairKV-LA* applies the leader-avoidance technique.
- *FlairKV-LA+FL* uses both leader-avoidance and follower load-awareness techniques.

The FlairKV-LA+FL awareness technique (6.2) helps in deployments with heterogeneous hardware and load variance between followers. To emulate a heterogeneous platform, we manually reduced the CPU frequency for one follower by 10%.

FlairKV-LA avoids the leader when it is busy serving write requests. To measure the efficiency of this approach we measure the system throughput with workload B that has 5% writes with uniform key popularity distribution. The results (Fig. 16.a) show that FlairKV-LA brings 40% higher throughput than FlairKV-Rand. This is due to avoiding the leader that becomes overloaded when receiving write requests. This reduces the load on the leader, consequently it accelerates write operations and reduces the period in which kgroups are marked unstable. FlairKV-LA+FL had comparable performance to FlairKV-LA as nodes are homogenous.

We run the same experiment on the emulated heterogeneous setup. Fig. 16.b shows the throughput of the different load balancing techniques. FlairKV-LA and FlairKV-LA+FL send majority of the reads to the two followers. Since one of the followers is slower the total system throughput is reduced. Nevertheless, FlairKV-LA+FL achieves 17% higher throughput than FlairKV-Rand and 3% higher throughput than FlairKV-LA. The negligible improvement over FlairKV-LA is because the high overhead of the write operations.

To eliminate the write operation overhead, we compared the systems' throughput with the read only workload C with a uniform distribution on the emulated heterogeneous setup (Fig. 16.d). FlairKV-LA+FL brings 17% higher throughput compared with the other two techniques, because it distributes the load proportionally to the node's request queue length. Furthermore, we noticed that FlairKV-LA+FL reduces latency by 10%. FlairKV-LA and FlairKV-Rand are equivalent under the read-only workload, because they distributed the load equally across the nodes. Fig. 16.c shows that under a read-only workload with a homogenous hardware, all load-balancing techniques achieve similar throughput.

7.6 Different Number of Keys

We compared the performance of FlairKV against FLeases using YCSB B workload (95:5 read:write ratio) while varying the number of keys in the system from 100K to 4M keys. Fig. 17 shows the throughput and the average latency with uniform and skewed workloads. For the uniform workload (Fig. 17.a and 17.c), FlairKV achieves up to 45% higher throughput and up to 51% lower latency compared to FLeases when using 4M keys. For the skewed workload (Fig. 17.b and Fig. 17.d), FlairKV achieved up to 21% higher throughput and up to 23% lower latency compared to FLeases when using 4M keys. We note that the performance of FlairKV and FLeases degrades slightly with larger data sets. For instance, FlairKV's has 10% lower throughput when storing 4M objects compared with when storing 100K objects. This performance degradation is due to Raft's implementation of the key value store and not due any changes in the level of contention on the object stability array.

8 Related Work

Network-accelerated systems. SwitchKV [22] uses SDN capabilities to route client requests to the caching node serving the key. A central controller populates the forwarding rules to invalidate routes for objects that are being modified and installs routes for newly cached objects. NetCache [21] proposes using the limited switch memory as a look-through cache. Due to the memory limitation NetCache prototype had 8 MB of cache in the switch. We note that the NetCache approach is orthogonal to FLAIR's. FLAIR can be further optimized by caching the most popular values in the switch.

Network-accelerated consensus. A number of recent efforts leverage SDN's capabilities to optimize consensus protocols. Speculative Paxos [33] builds a mostly ordered multicast primitive and uses it to optimize the multi-Paxos consensus protocol. Network-ordered Paxos (NOPaxos) [32] leverages modern network capabilities to order multicast messages and add a unique sequence number to every client request. NOPaxos uses these sequence number to serialize operations and to detect packet loss. Speculative Paxos and NOPaxos are optimized for operations that update the log but not for read operations. NetChain [43] and NetPaxos [44] implement replication protocols on a group of switches. These protocols are suitable for systems that store only a few megabytes of data (e.g., 8MB in the NetChain prototype). Unlike FLAIR, these efforts do not optimize for read operations. Reads are still served by the leader or a quorum of replicas. HovercRaft [45] is Raft-based protocol that offload the replication operation to a programmable switch. When the leader receives a read request, it can ask one of the followers to serve this read request. Hovercraft does not explore new consistency-aware load balancing techniques.

Consensus protocols optimized for the WAN. A number of consensus protocols are optimized for WAN deployments. Quorum leases [10] proposes giving a read lease to some of the followers. Mencius [46] is a multi-leader protocol in which each leader controls part of the log. EPaxos [47] is a leaderless protocol where clients can submit a request to any replica. Non-conflicting write can commit in one round trip, while conflicting writes will be resolved using Paxos. CURP [48] optimizes the write operation through exploiting commutativity between concurrent writes. In data center deployments, CURP reads are served by the leader and hence are limited to a single node performance, in WAN deployment CURP applies a technique similar to FLeases. Tempo [49] is a leaderless protocol that relies on timestamps to guarantee consistency. Each log entry is tagged with a timestamp and is considered committed only after all log entries with lower timestamps are committed. Delos [50] provides a virtual shared log with a convenient API. Applications are oblivious to the real implementation of the shared log, which can consist of multiple consensus instances with different implementations.

A number of recent protocols leverages RDMA to optimize consensus protocols. DARE [15] implements RAFT protocol over RDMA, and committing a write operation requires two RDMA write operations. APUS [51] is a Paxos-based consensus library that requires one RDMA write operation to commit an operations. Mu [52] is a microsecond scale consensus library. The three aforementioned protocols are leader-based and aim to utilize RDMA to optimize the replication of the log. Hermes [53] is leaderless protocol that uses logical timestamps to resolve write conflicts locally at each replica. A write operation is committed if replicated on all replicas. Hence, replicas can serve read operations locally. However, in a case of a replica or network failure, the system stalls until the failed replica is removed from the cluster.

9 Conclusion

We present FLAIR, a novel protocol that leverages the capabilities of the new generation of programmable switches to accelerate read operations without affecting writes or using leases. FLAIR identifies, at line rate, which replicas can serve a read request consistently, and implements a set of load-balancing techniques to distribute the load across consistent replicas. We detailed our experience building FlairKV and presented several techniques to cope with the restrictions of the current programmable switches. We hope our experience informs a new generation of systems that co-design network protocols with system operations.

Acknowledgment

We thank Lori Paniak for his help in running the experiments. This research was supported by an NSERC Discovery grant, NSERC Collaborative Research and Development grant, Canada Foundation for Innovation (CFI) grant, and a grant from Waterloo-Huawei Joint Innovation Lab.

References

[1] H. Attiya and J. Welch, Distributed Computing: Fundamentals, Simulations and Advanced Topics. John Wiley & Sons, Inc., 2004.
[2] L. Lamport, "Paxos made simple," ACM Sigact News, vol. 32, no. 4, pp. 18-25, 2001.
[3] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in Proceedings of the USENIX Annual Technical Conference, Philadelphia, PA, 2014: USENIX Association.
[4] F. P. Junqueira, B. C. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in Proceedings of IEEE/IFIP International Conference on Dependable Systems&Networks, 2011: IEEE Computer Society, doi: 10.1109/dsn.2011.5958223.
[5] B. Liskov and J. Cowling, "Viewstamped replication revisited," Technical Report MIT-CSAIL-TR-2012-021, MIT, 2012.
[6] J. Shute, R. Vingralek, B. Samwel et al., "F1: a distributed SQL database that scales," Proc. VLDB Endow., vol. 6, no. 11, pp. 1068-1079, 2013, doi: 10.14778/2536222.2536232.
[7] N. Bronson, Z. Amsden, G. Cabrera et al., "TAO: Facebook's distributed data store for the social graph," in Proceedings of the USENIX Technical Conference, San Jose, CA, 2013: USENIX Association.
[8] B. Atikoglu, Y. Xu, E. Frachtenberg et al., "Workload analysis of a large-scale key-value store," presented at the Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems, London, England, UK, 2012.

[9] C. Gray and D. Cheriton, "Leases: an efficient fault-tolerant mechanism for distributed file cache consistency," in Proceedings of the ACM Symposium on Operating Systems Principles (SOSP), 1989: ACM, doi: 10.1145/74850.74870.

[10] I. Moraru, D. G. Andersen, and M. Kaminsky, "Paxos Quorum Leases: Fast Reads Without Sacrificing Writes," in Proceedings of the ACM Symposium on Cloud Computing, Seattle, WA, USA, 2014: ACM, doi: 10.1145/2670979.2671001.

[11] J. Baker, C. Bond, J. C. Corbett et al., "Megastore: Providing scalable, highly available storage for interactive services," in Proceedings of the Conference on Innovative Data system Research (CIDR), 2011.

[12] G. DeCandia, D. Hastorun, M. Jampani et al., "Dynamo: amazon's highly available key-value store," in Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP), Stevenson, Washington, USA, 2007: ACM, doi: 10.1145/1294261.1294281.

[13] D. B. Terry, V. Prabhakaran, R. Kotla et al., "Consistency-based service level agreements for cloud storage," in Proceedings of the ACM Symposium on Operating Systems Principles (SOSP), Farminton, Pennsylvania, 2013: ACM, doi: 10.1145/2517349.2522731.

[14] B. F. Cooper, R. Ramakrishnan, U. Srivastava et al., "PNUTS: Yahoo!'s hosted data serving platform," Proc. VLDB Endow., vol. 1, no. 2, pp. 1277-1288, 2008, doi: 10.14778/1454159.1454167.

[15] M. Poke and T. Hoefler, "DARE: High-Performance State Machine Replication on RDMA Networks," in Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing, Portland, Oregon, USA, 2015, 2749267: ACM, pp. 107-118, doi: 10.1145/2749246.2749267.

[16] P. Hunt, M. Konar, F. P. Junqueira et al., "ZooKeeper: wait-free coordination for internet-scale systems," in Proceedings of the USENIX annual technical conference, Boston, MA, 2010.

[17] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: an engineering perspective," in Proceedings of the annual ACM symposium on Principles of distributed computing, Portland, Oregon, USA, 2007: ACM, doi: 10.1145/1281100.1281103.

[18] D. Mazieres, "Paxos made practical," ed, 2007.

[19] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," OSDI 2006.

[20] S. Al-Kiswany, S. Yang, A. C. Arpaci-Dusseau et al., "NICE: Network-Integrated Cluster-Efficient Storage," in Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing, Washington, DC, USA, 2017: ACM, doi: 10.1145/3078597.3078612.

[21] X. Jin, X. Li, H. Zhang et al., "NetCache: Balancing Key-Value Stores with Fast In-Network Caching," SOSP 2017.

[22] X. Li, R. Sethi, M. Kaminsky et al., "Be fast, cheap and in control with SwitchKV," NSDI 2016.

[23] "LogCabin storage system." https://logcabin.github.io (accessed April 14, 2019.

[24] "P4." https://p4.org (accessed April 14, 2019.

[25] H. Takruri, I. Kettaneh, A. Alquraan et al., "FLAIR: Accelerating Reads with Consistency-Aware Network Routing," in 17th USENIX Symposium on Networked Systems Design and Implementation, 2020.

[26] L. Lamport, "The part-time parliament," ACM Trans. Comput. Syst., vol. 16, no. 2, pp. 133-169, 1998, doi: 10.1145/279227.279229.

[27] "Barefoot Tofino." https://www.barefootnetworks.com/products/brief-tofino/ (accessed April 14, 2019.

[28] "Cavium / XPliant." https://origin-www.marvell.com/documents/netpxrx94dcdhk8sksbp/ (accessed April 14, 2019.

[29] "High-Capacity StrataXGS® Trident 3 Ethernet Switch Series." https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56870-series (accessed September 9, 2019.

[30] P. Bosshart, D. Daly, G. Gibb et al., "P4: programming protocol-independent packet processors," SIGCOMM Comput. Commun. Rev., vol. 44, no. 3, pp. 87-95, 2014, doi: 10.1145/2656877.2656890.

[31] N. McKeown, T. Anderson, H. Balakrishnan et al., "OpenFlow: enabling innovation in campus networks," SIGCOMM 2008

[32] J. Li, E. Michael, N. K. Sharma et al., "Just say no to paxos overhead: replacing consensus with network ordering," OSDI 2016.

[33] D. Ports, J. Li, V. Liu et al., "Designing distributed systems using approximate synchrony in data center networks," NSDI 2015

[34] B. M. Oki and B. H. Liskov, "Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems," presented at the Proceedings of the ACM Symposium on Principles of Distributed Computing, Toronto, Ontario, Canada, 1988.

[35] D. Mazieres, "Paxos made practical," Technical Report on http://www.scs.stanford.edu/~dm/home/papers/paxos.pdf, 2007.

[36] H. Takruri, I. Kettaneh, A. Alquraan et al., "Network-Accelerated Consensus for Read-Intensive Workloads " University of Waterloo, 2021. [Online]. Available: https://cs.uwaterloo.ca/~alkiswan/papers/FLAIR-TR.pdf

[37] "The Raft Consensus Algorithm." https://raft.github.io/ (accessed April 14, 2019.

[38] "Barefoot P4 Studio." https://www.barefootnetworks.com/products/brief-p4-studio/ (accessed April 14, 2019.

[39] "P4 v16 Portable Switch Architecture (PSA)." https://p4.org/p4-spec/docs/PSA-v1.0.0.html (accessed April 14, 2019.

[40] "NOPaxos consensus protocol." https://github.com/UWSysLab/NOPaxos (accessed April 14, 2019.

[41] L. Lamport, "Fast paxos," Distributed Computing, vol. 19, no. 2, pp. 79-103, 2006.

[42] "Yahoo! Cloud Serving Benchmark in C++, a C++ version of YCSB." https://github.com/basicthinker/YCSB-C (accessed April 14, 2019.

[43] X. Jin, X. Li, H. Zhang et al., "Netchain: scale-free sub-RTT coordination," NSDI 2018.

[44] H. T. Dang, D. Sciascia, M. Canini et al., "NetPaxos: consensus at network speed," ACM SIGCOMM 2015.

[45] M. Kogias and E. Bugnion, "HovercRaft: achieving scalability and fault-tolerance for microsecond-scale datacenter services.," EuroSys 2020.

[46] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: building efficient replicated state machines for WANs," OSDI 2008.

[47] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in Egalitarian parliaments," SOSP 2013.

[48] S. J. Park and J. Ousterhout, "Exploiting commutativity for practical fast replication," NSDI 2019.

[49] V. Enes, C. Baquero, A. Gotsman et al., "Efficient Replication via Timestamp Stability," in Proceedings of the Sixteenth European Conference on Computer Systems, 2021.

[50] M. Balakrishnan, J. Flinn, C. Shen et al., "Virtual Consensus in Delos," in 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), 2020.

[51] C. Wang, J. Jiang, X. Chen et al., "APUS: Fast and Scalable Paxos on RDMA," in Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17), 2017.

[52] M. K. Aguilera, N. Ben-David, R. Guerraoui et al., "Microsecond Consensus for Microsecond Applications," OSDI 2020.

[53] A. Katsarakis, V. Gavrielatos, M. R. S. Katebzadeh et al., "Hermes: A Fast, Fault-Tolerant and Linearizable Replication Protocol," ASPLOS 2020.