Network-Accelerated Linearizable Reads

by

Hatem Takruri

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2019

AUTHOR'S DECLARATION

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

This work was done in collaboration with Ibrahim Kettaneh and Ahmed Alquraan under the supervision of Prof. Samer Al-Kiswany from the University of Waterloo, Ibrahim Kettaneh helped in implementing the P4 pipeline (Section 6.2), and Ahmed Alquraan helped in writing the TLA+ proof specification (Appendix B).

Abstract

This thesis 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. The evaluation indicates that the proposed approach can bring significant performance gains: up to 43% higher throughput and 35-97% lower latency for most workloads compared to state-of-the-art alternatives.

Acknowledgements

I would like to express my deep gratitude to my advisor Prof. Samer Al-Kiswany for his relentless support, guidance, and advice. This work would not be possible without him.

Many thanks to my colleagues Ibrahim Kettaneh and Ahmed Alquraan for their tremendous help and support throughout this project.

I would like finally to thank my parents and brothers for their love, support and encouragement.

Table of Contents

List of Figures

Chapter 1
Introduction

Replication is the main reliability technique for many modern cloud services [1, 2, 3] that process billions of requests each day [3, 4, 5]. Unfortunately, modern strongly-consistent replication protocols [6] – such as multi-Paxos [7], Raft [8], Zab [9], and Viewstamped replication (VR) [10] – 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 [11] and 500:1 in Facebook's TAO [5]. Previous efforts have attempted to accelerate reads by giving read leases [12] to some (quorum leases [13]) or all followers (Megastore [1]). 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, affects the write operation – as all granted leases need to be revoked before an object can be modified – and imposes long delays when a follower holding a lease fails [1, 13].

Alternatively, many systems support a relaxed consistency model, such as eventual [2, 14, 15, 16, 17, 18] or read-your-write consistency [5, 18, 19], in exchange for the ability to read from followers, albeit the possibility of reading stale data.

In this thesis, 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 techniques.

FLAIR is positioned as a shim layer on top of a leader-based consensus protocol (Chapter 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 [8], VR [10], DARE [20], Zookeeper [2], and multi-Paxos implementations [21, 22, 23]).

FLAIR leverages the power and flexibility of the new generation of programmable switches. The core of FLAIR is a packet-processing pipeline (Chapter 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 from followers 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 consistency-aware load balancing techniques. In systems that grant a lease to a follower [1, 13, 24], 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 balances read requests only among followers that hold a consistent value for the requested object. In this thesis we explore the design of three consistency-aware load balancing techniques (Chapter 6): random, leader avoidance, and load awareness.

Unlike other systems that use switch's new capabilities [25, 26, 27], 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.

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

To demonstrate the powerful capabilities of the proposed approach, we prototyped FlairKV (Chapter 6), a key-value store built atop Raft [8]. We made only minor changes to Raft's implementation [28] 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 [29]. We implemented the three aforementioned load-balancing techniques (Section 6.2)

The evaluation of FlairKV (Chapter 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. Using the YCSB [30] benchmark with different read-to-write ratios, FlairKV achieves 1.2 to 2.5 times higher throughput than an optimized Raft implementation, at least 7 times higher throughput compared to Viewstamped replication, and up to 43% higher throughput and up to 35-97% lower latency for most workloads compared to state-of-the-art leases-based design [1, 24].

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 the experience (Chapter 6) will inform a new generation of distributed systems that co-design network protocols with systems operations.

Chapter 2
Background and Related Work

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

2.1 Leader-based Consensus

Leader-based consensus (LC) protocols [8, 9, 10, 20, 21, 22] are widely adopted in modern systems [2, 3, 4, 24]. The idea of having a leader that can commit an operation in a single round trip dates back to the early consensus protocols [7, 31]. Having a leader reduces contention and the number of protocol messages, which greatly improves performance [7, 21].

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 Figure 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 key-value store (6), then acknowledges the operation to the client (7). The leader will asynchronously inform the followers that the operation has committed. 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.

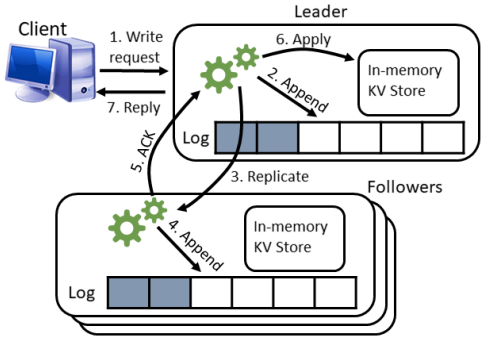


Figure 1. The path for a write operation.

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. These properties hold for all major protocols (Raft [8], VR [10], Zab [9], DARE [20], and multi-Paxos implementations [22]).

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 before serving the request.

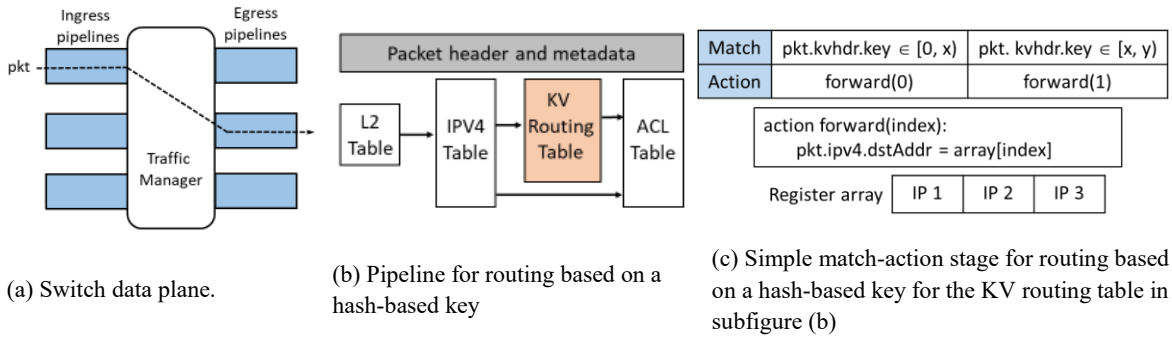
A common optimization is the leader lease optimization. Instead of collecting a majority of heartbeats for every read request, a majority of the followers can give the leader a lease [8, 21]. While holding a lease, the leader serves reads locally without contacting followers. Unfortunately, even with this optimization, leader-based protocols' performance is limited to a single-node performance.

2.2 Programmable Switches

Software-defined networking (SDN) divides the network into two planes: data and control. The data plane is a packet processing and forwarding plane. The control plane is an external software-based component that controls one or more 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 [32] and Cavium's XPliant [33].

Figure 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.



(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)

Figure 2. Switch data plane.

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 per-packet headers and metadata as well as custom actions. Each stage has its own dedicated resources, including tables and register arrays (a memory buffer). Figure 2(b) shows a simple example of a pipeline that routes a request to a key-value store based on the key, and Figure 2(c) shows the details of the KV routing stage in Figure 2(b). 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 entries in the table.

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 (Figure 2(b)). The processing of packets can be viewed as a graph of match-action stages.

Programmers use domain-specific languages like P4 [34] 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, and (6) lack support of loops or recursion.

2.3 Related Work

Network-accelerated systems. Recent projects have utilized SDN capabilities to provide load balancing [35, 36, 37], access control [38], seamless virtual machine migration [39], and improving system security, virtualization, and network efficiency [40]. SwitchKV [27] 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 [26] implements a caching service in a single switch. The controller keeps track of the most popular objects and controls the cached objects in the switch.

Network-accelerated consensus. A number of recent efforts leverage SDN's capabilities to optimize consensus protocols. Speculative Paxos [41] builds a mostly ordered multicast primitive and uses it to optimize the multi-Paxos consensus protocol. Network-ordered Paxos (NOPaxos) [42] 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] optimizes vertical Paxos [44] by implementing chain replication on a chain of programmable switches. NetPaxos [45] considers moving the Paxos protocol to the network switches, such that one switch serves as a coordinator and other switches serve as replicas. The proposed approach requires implementing a substantial part of the protocol in switches and storing a potentially large protocol state. NetChain and NetPaxos are suitable for systems that store only a few megabytes of data (e.g., 8MB in the current NetChain prototype).

Consensus protocols optimized for the WAN. A number of consensus protocols are optimized for WAN deployments. Quorum leases [13] proposes giving a read lease to some of the followers; Unlike Megastore leases, when an object is modified, only the followers that have the lease are contacted. Quorum leases achieves better performance than Megastore leases in WAN setups, but do not bring benefits when deployed in a single cluster [13]. Mencius [46] is a multi-leader state machine replication protocol that utilizes a rotating coordinator scheme to assign consensus log instances different servers. This allows all servers in the replica-set to share the load. Each client sends a request to the closest server, and the server processes the request in its entries in the log. EPaxos [47] is a leaderless protocol where clients can submit request to any replica. Non-conflicting write can commit in one round trip, while conflicting writes will be resolved using Paxos.

Chapter 3
Flair Overview

FLAIR is a novel protocol that targets deployments in a single data center. Figure 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).

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. Finally, FLAIR assumes a fail-stop model in which nodes and switches may stop working but will never send erroneous messages. FLAIR guarantees per-object linearizability, it does not support multi-object transactions.

FLAIR divides time into sessions (Figure 4). A session represents a binding between a leader and the switch for a period of time. Each session has a unique id that is assigned in a strictly increasing order. An LC term may have one or more sessions, but a session does not span multiple terms. A session ends when a leader fails or the *lflair* module suspects that the switch has failed.

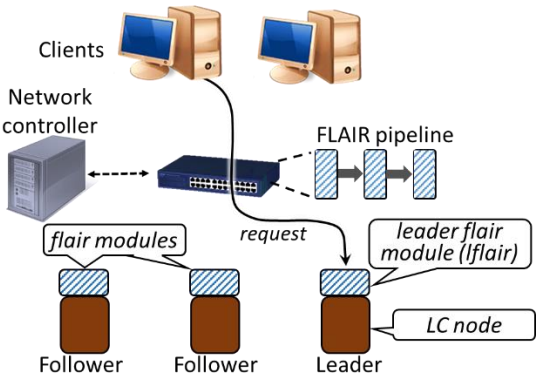


Figure 3. System architecture. The solid arrow shows a client request, while the dashed arrow show control messages.

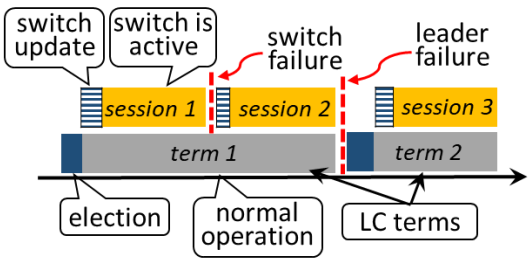


Figure 4. FLAIR sessions. Time is divided into terms. Each term starts with a leader election. Each term has one or more sessions that start with updating the switch data.

A session starts with the FLAIR module at the leader (dubbed the *lflair* module) incrementing the session id, committing it to the LC log, updating the switch information about the objects in the system, then activating the session at the switch. *lflair* module keeps the switch's information up to date while in an active session. If the switch does not have an active session it drops all FLAIR packets.

Clients. FLAIR is accessed through a client library with a simple interface for read, write, and delete operations. 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. FLAIR design targets data centers that use a SDN network following a variant of the multi-rooted tree topology [48, 49]. A central controller uses OpenFlow [50] to manage the network by installing per-flow forwarding, filtering, and rewriting rules in switches.

As with previous projects that leverage SDN capabilities [25, 27, 41, 42], 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; 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 network forwarding, the effect of this change is minimal in practice. Li et al. [42] 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.

On a switch failure, the controller selects a new switch and updates all the forwarding rules accordingly. The controller load balances the work across switches by assigning different replica sets to different switches.

Storage Nodes. The storage nodes run the FLAIR and LC protocols. 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 *lflair* module the log index at which the write was committed and which followers have a consistent log up to that log index. The *lflair* encodes this list into a compact bitmap and uploads it and the log 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 (not being modified) and which replicas have a consistent value of each object (encoded in the bitmap provided by the *lfair* module). 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 *lflair* module. Followers optimistically serve read requests. The switch inspects every read reply; if it suspects that a follower returned stale data (Section 4.5), 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 (Section 4.4), packets reordering (Section 4.6), and tolerating switch, node, and network failures (Section 4.6).

Chapter 4
System Design

4.1 Network Protocol

Packet format. FLAIR introduces an application-layer protocol embedded in the L4 payload of packets. Similar to many other storage systems [25, 27, 42], 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.

Figure 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 operation.
- 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 indicates 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.

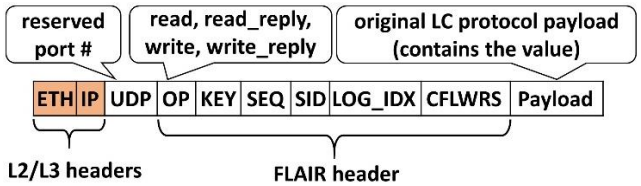


Figure 5. FLAIR packet format.

- 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 operations.

4.2 Switch Data Structures

To process a read request, the switch performs two specific tasks (Section 4.5). First, it forwards read requests to consistent followers while balancing the load across followers. 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). Each replica set stores a disjoint set of objects. Each entry in the session array maintains the session status for a single replica set. Each entry (Listing 1) contains an is_active flag, current session sequence number, leader IP address, session id, and the timestamp of the last heartbeat received from the *lflair* module. 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).

```
SessionArrayEntry {
  bit<1>  is_active;
  bit<64> session_seq_num;
  bit<32> leader_ip;
  bit<32> session_id;
  bit<48> heartbeat_tstamp;
}
```

```
KGroupArrayEntry {
  bit<1>  is_stable;
  bit<64> seq_num;
  bit<64> log_idx;
  bit<8>  consistent_followers;
}
```

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

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 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. 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 starts from zero.

To populate 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).

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. 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 session (sets is_active to true).

4.4 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 packet 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 (Section 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) using the SEQ of the 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 to the kgroup (with a SEQ number equal to the seq_num in the kgroup entry) is acknowledged by the leader.

4.5 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. Section 6.2 details the 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 in the read request. If the commit_index is smaller, the follower will advance 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 Chapter 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 a stale value. In this case, the switch drops the read_reply and resubmits the read request to the leader. If the sequence number of the read_reply equals the sequance 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, without checking the seq_num in the kgroup entry.

4.6 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 (Chpter 6).

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 (Section 4.3).

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 the switch heartbeats for a switch_stepdown period of time (three heartbeats in the prototype), the *lflair* module will suspect that the switch has failed and will start a new session (Section 4.3). 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, affecting performance but not correctness.

Packet Reordering. It is critical for FLAIR 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 are modifying its objects. To prevent this scenario, the leader keeps track of the largest <SID, SEQ> combination 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, and resets the session_seq_num to zero.

Chapter 5
Correctness

FLAIR guarantees linearizability, which means that concurrent operations must appear to be executed by a single machine. FLAIR relies on the LC layer for any operation that updates the log and for reads from the leader.

FLAIR only adds the ability to serve reads from followers. In this chapter, we sketch out the proof of FLAIR correctness when the read is served by a follower. A full and detailed proof is available in Appendix A. Further, we used the TLA+ model checking tool [41] to verify the FLAIR correctness. We started from Raft's TLA+ specification [39] and extended it with a formal specification for the protocol and new invariants to validate the linearizability of reads. The TLA+ specification is available in Appendix B.

Safety. FLAIR guarantees that all read replies are linearizable. FLAIR trusts that the leader's read replies are linearizable and forwards them to the client. For reads served by followers, FLAIR guarantees that the read reply returns an identical value, as if the read was served by the leader. This is guaranteed using the following two steps:

First, when the switch receives a read request, the switch forwards that request to followers only when the switch has an active session and the kgroup entry is stable. This signifies that the switch information is up-to-date with the *lflair* module's information. Identifying a kgroup entry as stable signifies that there are no current writes to any object in the kgroup and that the last leader-provided consistent_followers bitmap points to followers that have the last committed value for every object in the kgroup. Consequently, any of the consistent followers will return a value identical to the leader's value.

Second, after forwarding a read request to a follower (say, flwrA), 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 flwrA. If the leader processes the write request before flwrA serves the read request, flwrA will return stale data. To avoid this case, 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 conservatively drops the read reply and forwards the request to the leader.

At all times, reads are linearizable in FLAIR.

Chapter 6
Implementation

To demonstrate the benefits of the new approach, we prototyped FlairKV, a FLAIR-based key-value store built atop Raft [28]. We chose Raft due to its adoption in production systems [51, 52, 53, 54, 55], and the availability of standalone production-quality implementations [56].

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 [8, 21] 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 [29] and is compiled for Barefoot's Tofino ASIC [32], with Barefoot's P4Studio software suite [57]. The prototype 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. In total, the prototype 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 chapter 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_stepdown period passes without receiving a leader heartbeat, 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 (Section 6.3). 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 switch_stepdown, 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 (Section 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; Figure 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" [58].

Load balancing. In addition to the aforementioned random load-balancing technique (Figure 6), we implemented two load-aware techniques:

- *Leader avoidance*. The prototype 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.

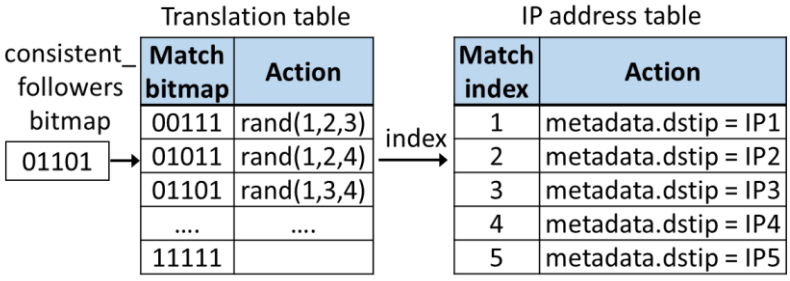


Figure 6. Logical view of the forwarding logic. The stability bitmap matches an entry in the translation table and executes the corresponding action, generating an index of the selected destination's IP address. Using the index, the IP address table sets the destination's IP address in the metadata.

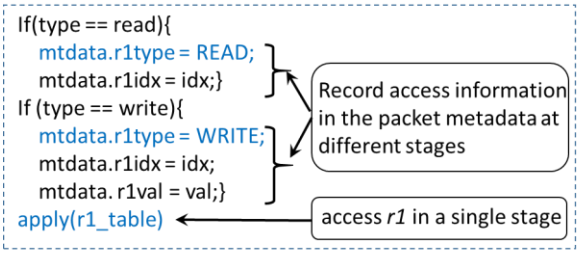
To implement this technique, we extended the session array entry with a 64-bit largest_reply_seq_num field, which tracks the largest sequence number ever reported in a write_reply. If largest_reply_seq_num does not equal session_seq_num, then there are pending writes in the system and the leader should 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., replicas for different ranges of keys) on the same machine.

  In the prototype design, 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 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.

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. Figure 7 shows an example of an action table for accessing register *r1*. The 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. Then a dedicated match-action table (Figure 7) is used 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 [58] to the FLAIR pipeline.



Match	Action
mtdata.r1type == READ	mtdata.r1val = reg1[mtdata.r1idx]
mtdata.r1type == WRITE	reg1[mtdata.r1idx] = mtdata.r1val

Register r1 access table (**r1_table**)

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

6.3 Putting the Switch Pipeline Together

Figure 8 shows the pipeline layout in the switch data plane and the flow for a FlairKV packet. The pipeline starts by reading the session information (1 in Figure 8) and adding it to the packet metadata. Subsequently, the operation type is extracted (2) and the request is validated (3) through verifying packet's session id. If the packet has an older session id the packet is dropped. Further, in the validation stage the switch confirms that it did not miss leader heartbeats in the last switch_stepdown period (Section 4.6), else it deactivates the session.

Read requests access the kgroup array (6), and if the group is stable, the request is forwarded to a load-balancing logic (10) that implements the forwarding logic (Section 6.2); otherwise, it is sent to the leader.

If a read reply is from the leader, it is forwarded to the client (12). If it is from a follower, the pipeline performs the safety check (9) and, if it suspects the reply is stale, drops the reply, then resubmits the read request to the leader (11).

Write requests update the session_seq_num (4) and the kgroup entry (6), then are sent to the leader (11).

Write replies compare the sequence number of the reply to the one in the kgroup entry (5); if they match, the kgroup entry is updated (6) and the pipeline forwards the reply to the client (12).

The egress pipeline (13) has one logical stage that populates the header fields (e.g., SEQ number, SID, etc.) using the data available in the packet's metadata.

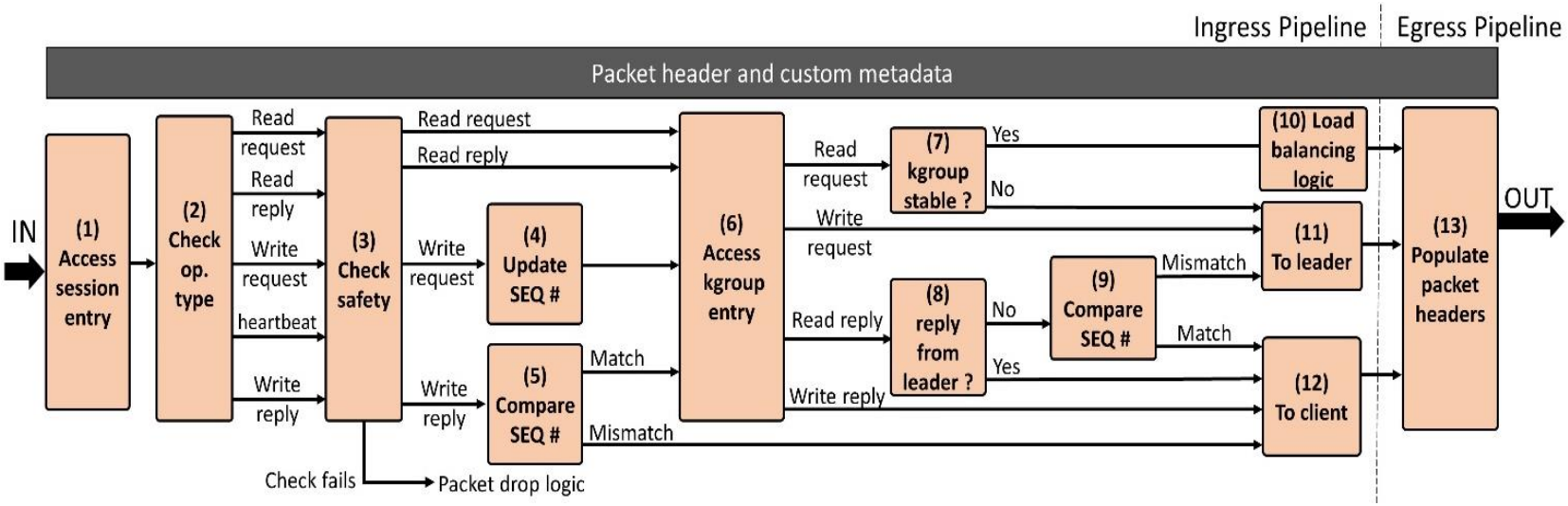


Figure 8. Logical view of the FlairKV switch data plane.

Chapter 7
Evaluation

We compare FlairKV prototype with previous approaches in terms of throughput and latency (Section 7.1) and evaluate the benefits of different load-balancing techniques (Section 7.2). Finally, we demonstrate FlairKV's fault-tolerance techniques (Section 7.3).

Testbed. We conducted the experiments using a 13-node cluster. Each node has an Intel Xeon Silver 10-core CPU, 48GB of RAM, and 100Gbps Mellanox NIC. The nodes are connected to an Edgecore Wedge 100 ×32BF switch with 32 100Gbps ports. The switch has Barefoot's Tofino ASIC, which is P4 programmable. In all the experiments, 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, the original implementation of Raft (Raft), and an implementation of Viewstamped replication (VR) [23]. Raft and VR implement a batching optimization which batches and replicates multiple log entries in a single round trip.
- **Optimized Leader-based (Opt. Raft)**. Throughput benchmarking revealed that the original Raft implementation could not utilize the resources of the cluster. Two main optimizations were implemented: first, we changed the request-processing logic from an event-driven to a thread-pool design, as the conducted benchmarking indicated a thread-pool performs better; second, we implemented the leader-lease optimization. These changes significantly improved Raft's performance.
- **Quorum-based reads (Fast Paxos)**. An alternative to the leader-based design is the quorum design **[41, 42, 59]**. Typically, 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 **[59]** before completing a read. We used a Fast Paxos implementation [23] that implements only the normal case.
- **Follower-lease optimization (FLeases)**. Similar MegaStore [1] (also explored in Chubby [21]), 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. If a follower receives a read request for an object for which it did not have an active lease, it forwards the request to the leader. 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.

- **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. For all experiments, we stored all logs in memory.

Workload. We used the YCSB benchmark [30] to evaluate the performance of all systems. We considered both uniform and skewed workloads. The skewed workload follows the Zipf distribution with a skewness parameter of 0.99. We used three YCSB workloads: workload A has a 1:1 read-to-write ratio; workload B has a read-to-write ratio of 95:5, and workload C is read-only.

The 10 client nodes each ran 100 threads, each thread continuously generated read and write requests in a closed loop. For workload A some of the systems did not sustain this many clients and we had to reduce the number of threads to 20 per client, but that still generated enough load to stress all the systems. We used 100,000 keys, with a key size of 24 bytes. The hash of the key string is used as the key in the FLAIR protocol. The value size is 1KB.

7.1 Performance Evaluation

Throughput. Figure 9 and Figure 10 show the throughput of the six systems for workloads A, B, and C. For workload C under both the uniform (Figure 9.C) and Zipf (Figure 10.C) distributions, FlairKV and FLeases had the highest throughput, 2.7 M op/s, as both systems can utilize all replicas to serve read requests. FlairKV and FLeases achieved 2.8 times higher throughput relative to OptRaft, which only uses the leader to serve requests. Finally, FlairKV and FLeases had at least 42 times higher

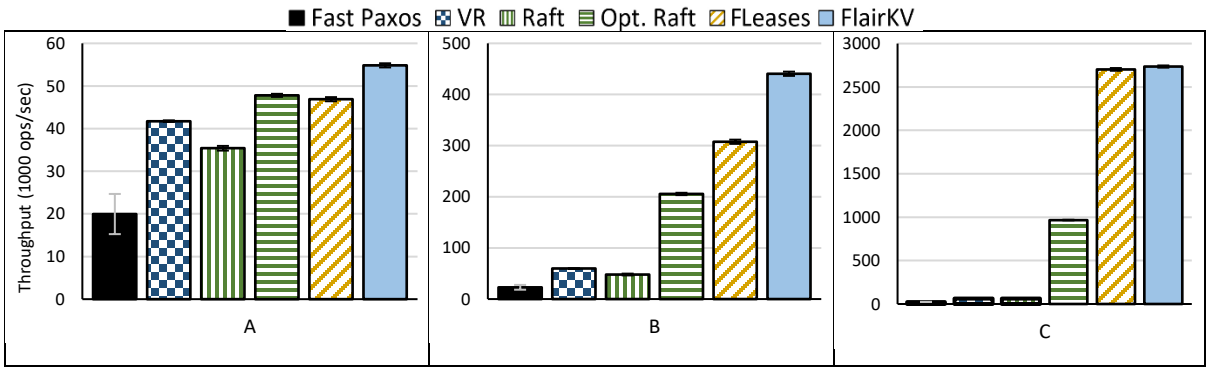


Figure 9. System's throughput (uniform workload) using the YCSB workloads A, B, and C with uniform key popularity distribution. Error bars show standard deviation, which is less than 1% for all systems except Fast Paxos, which had higher variance.

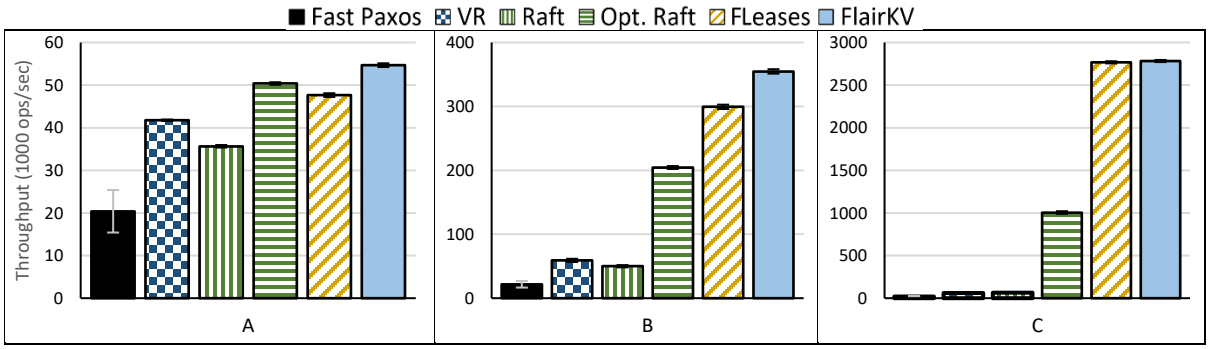


Figure 10. System's throughput (Zipf workload) using the YCSB workloads A, B, and C with Zipf key popularity distribution. Error bars show standard deviation which is less than 1% for all systems except Fast Paxos, which had higher variance.

throughput relative to Raft, VR, and Fast Paxos, as these protocols contact a majority of followers for every read operation.

Uniform distribution. Figure 9.B shows the throughput under workload B, with FlairKV achieving the highest performance. We noticed a significant reduction in throughput in FlairKV, FLeases, and OptRaft relative to workload C. The benchmarking revealed that this is due to the high overhead of the write operation; it takes 35 times longer to process a write relative to a read in OptRaft, and, for FlairKV and FLeases, due to writes marking kgroups unstable or revoking leases.

FlairKV achieved 43% higher throughput than FLeases for three primary 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. We recorded the number of read requests served by the leader and found that it only served 2% of the read requests in FlairKV (those are reads to unstable kgroups), while it served 34% of the reads in FLeases.

Second, when an object is not stable, if a client sends a request to a follower, the follower will redirect it to the leader, incurring extra latency. On the other hand, FlairKV switch knows if an object is not stable and forwards read requests for that object directly to the leader; Third, FLeases write operations need to reach all the followers, while FlairKV writes only need a majority. FlairKV had a 2.1 times higher throughput than OptRaft, and at least 7.4 times higher than Raft, VR, and Fast Paxos.

Figure 9.A shows the throughput under write-intensive workload A. FlairKV had the highest performance; 16% higher than FLeases and OptRaft, and around 31% and 54% higher performance than VR and Raft, respectively. Fast Paxos has the lowest throughput.

We note that the performances of Raft, VR, and Fast Paxos do not change significantly across the workloads, as reads still involve a majority of the followers.

Skewed distribution. Under the Zipf popularity distribution (Figure 10), FlairKV had a comparable performance improvement, with a slight reduction in throughput under workloads A and B due to increased contention on the popular keys.

We noticed that for the write-heavy workload A, FlairKV improved throughput by only 16% over FLeases. The first reason behind this performance is that writes dominate the system performance and both systems use the same write path. But the second, and a subtle reason, is due to a side effect of FLAIR. When there are concurrent writes to the same kgroup, FLAIR will mark a group unstable from moment the first request is processed by the switch until the last request to the kgroup is replied to. For example, in Figure 11 the kgroup is marked unstable for the entire period [*t1*, *t2*]. In FLeases, the lease revocation is piggybacked on the write replication step (black diamonds in Figure 11). Once the leader commits a write, it sends a commit notification and grants a new lease to the followers (white diamonds). Figure 11 shows an example in which FLeases can grant a lease between concurrent writes, therefore creating more opportunity for serving reads from followers.

To further understand this effect, we tracked leases and the stability of kgroups under the skewed workload A. We noticed that while 29% of reads found the kgroup unstable in FlairKV, while only 4% of reads in FLeases reached a follower that did not have a lease. We further profiled the write operation path and found that FLeases revokes leases for 70% of the write operation time (Figure 11), 30% shorter than the period FlairKV marks a kgroup unstable. Despite this subtle effect FlairKV leader still has lighter load. FlairKV leader served 29% of reads, while FLeases leader served 37% of

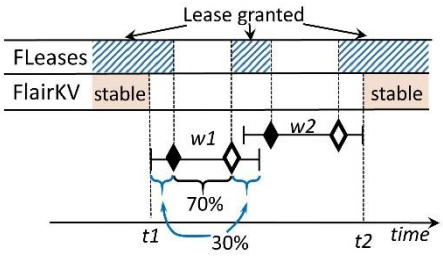


Figure 11. Subtle effect of FLAIR. w1 and w2 are write operations to the same kgroup. Bars mark the time from the moment a switch receives a write request until it receives a corresponding reply. For the FLeases, black diamonds mark when a leader replicates a write and revokes follower leases, and white diamond mark when a write is committed and the lease is granted to all followers.

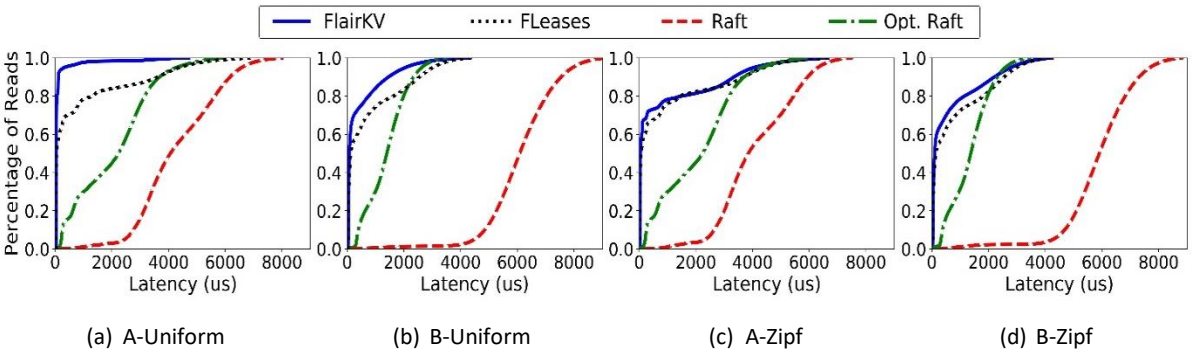


Figure 12. Latency CDF. The figures shows the latency CDF for reads using the uniform distribution under workload A (a) and B (b), and using the Zipf distribution under workload A (c) and B (d).

reads (33% due to clients picking the leader at random, plus 4% redirected writes). Notwithstanding this effect FlairKV still brought 16% performance improvement even under skewed write heavy load.

Latency evaluation. Figure 12 shows the latency of FlairKV, FLeases, OptRaft, and Raft. Under the uniform distribution (Figure 12 (a and b)), FlairKV lowered the latency for the slowest 40% requests by at least 38% relative to FLeases.

Under the Zipf workload, for workload B (Figure 12 (d)), FlairKV achieved up to 35% lower latency relative to FLeases. For write-heavy workload A (Figure 12 (c)) FlairKV and FLeases had a comparable latency.

FLease 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 OptRaft and Raft. The median latency of FlairKV is 2% of Raft's latency and 2-8% of OptRaft's latency. For workload C (not shown), FlairKV and FLeases had comparable latency.

Summary. The evaluation shows that FLAIR can yield measurable throughput and latency improvements, even under write-heavy workloads: up to 43% throughput improvement and up to 35-97% latency improvement relative to the state-of-the-art read optimization technique. These results demonstrate the significant benefits of co-designing packet processing with system protocols to realize more efficient, scalable, and reliable distributed systems.

7.2 Load-balancing Performance Evaluation

We measured the system throughput using the following three configurations of FlairKV (detailed in Section 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.

Figure 13.a shows the performance improvement produced by the leader-avoidance technique. In this experiment, we used workload B with uniform key popularity distribution. The results show that FlairKV-LA throughput is higher by 40% than FlairKV-Rand throughput, as it accelerates writes and reduces the period in which kgroups are marked unstable. FlairKV-LA+FL had comparable performance to FlairKV-LA as nodes are homogenous.

Figure 13.b evaluates the benefits of using the follower-load awareness technique (Section 6.2). This technique helps in deployments with heterogeneous hardware and with load variance. To emulate such scenarios, we manually reduced the CPU frequency for one follower by 10%. We used the read-only workload C with uniform distribution to avoid write operations (as those give advantage to the leader-avoidance technique). FlairKV-LA+FL had 17% higher throughput relative to the other configurations, as 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.

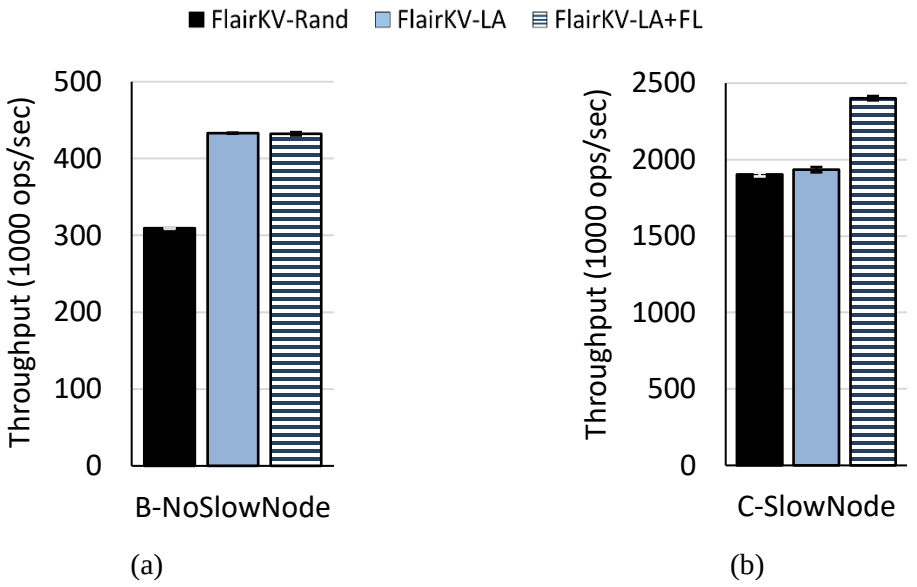


Figure 13. Throughput using different load-balancing techniques. (a) Uses workload B without slowing any follower and (b) uses workload C and slows one of the followers.

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 (Figure 14). 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. Figure 15 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 (Figure 16). At the 10s mark, we kill a follower process. This causes a drop in throughput as fewer replicas are available to serve read

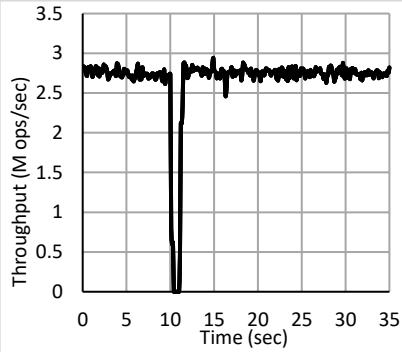


Figure 14. FlairKV throughput during a switch failover.

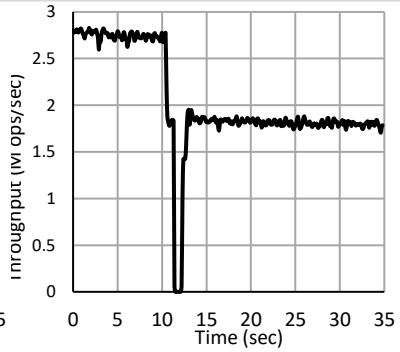


Figure 15. FlairKV throughput during leader failover.

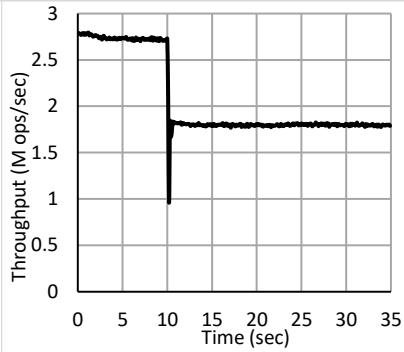


Figure 16. FlairKV throughput during a follower failure.

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 Workload Skewness

We measured the impact of the Zipfian workload skewness on the throughput as shown in Figure 17. Skewness effect by varying the Zipfian constant parameter θ from 0.5 to 0.99. With very high skewness (0.99), FalirKV loses its performance as more keys are now contenting on fewer KGroups in the switch rendering it to be unstable for longer periods, hence, read operations are forwarded to the leader and the followers are less utilized. Other systems do not experience much loss on their throughput as they already have contention. FLeases for instance has to revoke and grant leases for every write operation. Opt. Raft, Raft, and FastPaxos have the leader fully loaded, thus increasing the skewness will not affect their performance.

7.5 Varying Read Percentages

Figure 18. Different read percentages shows the effect of different percentages of reads operations on the throughput. Fast Paxos and Raft show limited throughput as both these systems have to reach majority of nodes to serve read operations. Opt. Raft, Unreplicated, FLeases, and FlAIR demonstrated throughput improvement when increasing the reads percentage as they have optimizations for serving read operations. FLAIR shows the highest throughput among all systems. Unreplicated and FLeases come next, with FLeases being more optimized for read-heavy workloads as the follower being utilized.

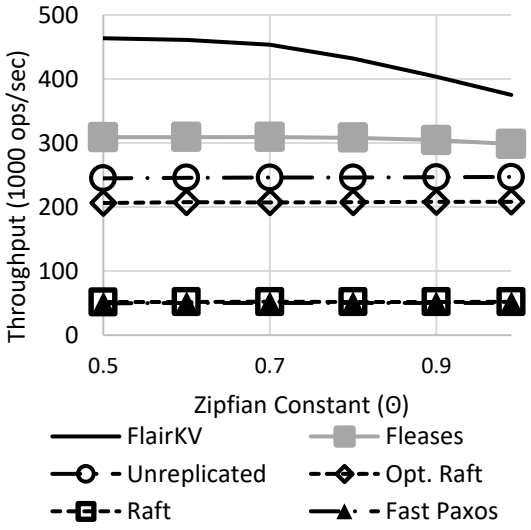


Figure 17. Skewness effect

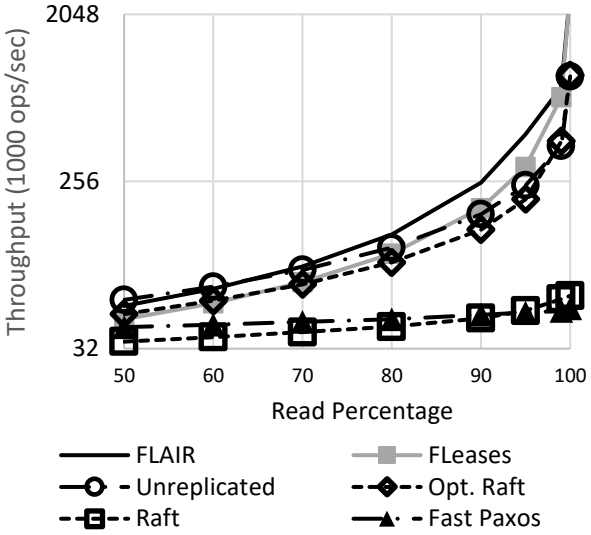


Figure 18. Different read percentages

7.6 Throughput-Latency Evaluation

Figure 19. Throughput vs clients shows the throughput of FALIR and FLeases with varying the number of clients under uniform distribution of workload B. FLAIR reaches its maximum throughput at 300 clients reaching 450,000 op/sec, however by increasing the number of clients beyond 300, the system is handling more operations rendering the KGroup entries to be frequently unstable, thus, operations are forwarded to the leader which result in slightly decrease of throughout. Similarly, FLeases reaches its maximum throughput at 180 clients after that the thorough decreases slightly as more operation get forwarded to the leader from followers which do not have a proper lease. Opt. Raft and Unreplicated show stable throughput as only the leader being fully utilized. Fast Paxos and Raft both are unable to handle large number of clients as each operation requires majority of nodes, both systems will start to drop and timeout operations.

Using the same setup, Figure 20. Read latency vs clients shows the read latency for FLAIR and FLeases, both systems have comparable latency with increasing the number of clients until the point they reach their maximum throughput. Increasing the number of clients beyond their maximum throughput, FLAIR can server reads ~200us faster than FLeases. Raft and Fast Paxos are unable to handle large number of clients.

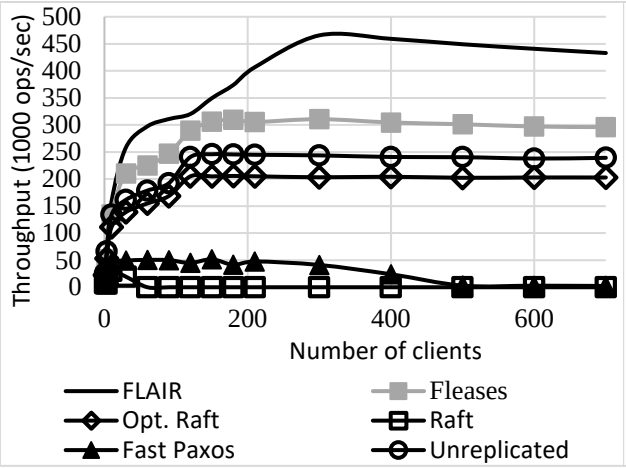


Figure 19. Throughput vs clients

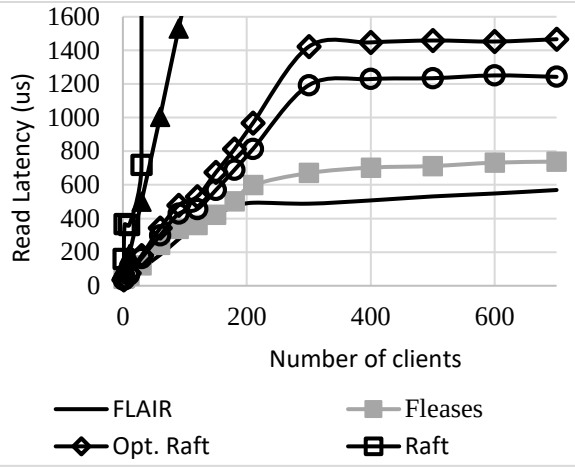


Figure 20. Read latency vs clients

7.7 Additional Evaluation

This section presents additional evaluation results. In the following experiments we use the same setup, run the same workloads. The only difference here is that we use 5 nodes as servers and 8 servers to generate client workload.

Figure 21 shows the throughput of the six systems for workloads A, B, and C. For workload C (Figure 21.C), FlairKV and Leases achieve the highest throughput, 4.4 M op/s, as both systems can utilize all replicas to serve read requests. FlairKV and Leases achieve 4.7 times higher throughput relative to OptRaft, which only uses the leader to serve read requests. Finally, FlairKV and Leases achieve at least 82 times higher throughput relative to Raft, VR, and Fast Paxos, as these protocols contact the majority of followers for every read operation.

Figure 21.B shows the throughput under workload B, with FlairKV achieving the highest performance. FlairKV achieves 46% higher throughput than Leases for three primary reasons. First, FlairKV uses the leader-avoidance load-balancing technique, which accelerates writes and reduces the time in which kgroups are marked unstable. We recorded the number of read requests served by the leader and found that it only served 2.9% of the read requests in FlairKV (those are reads to unstable kgroups), while it served 20% of the reads in Leases.

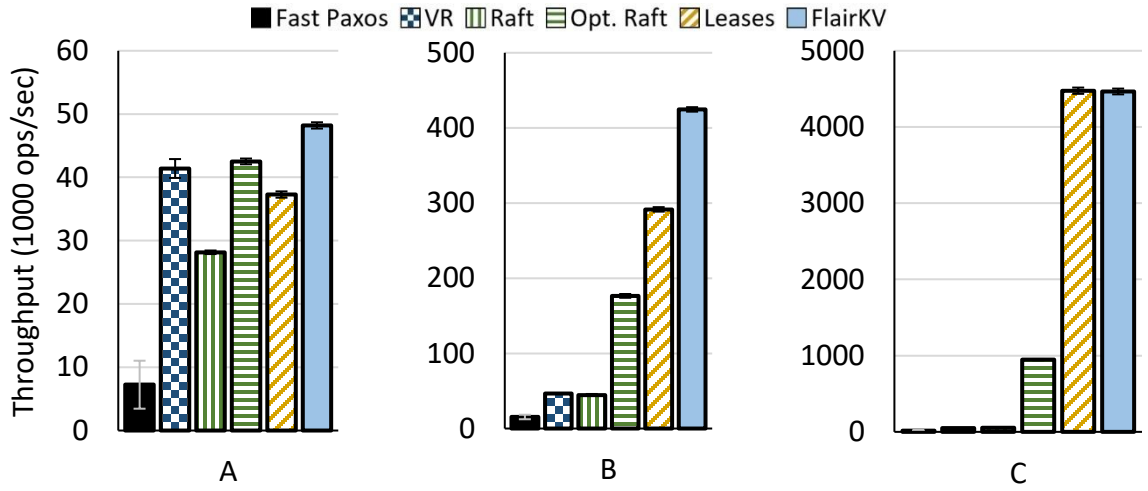


Figure 21. System's throughput (5 replicas, uniform workload) using the YCSB workloads A, B, and C with uniform key popularity distribution. Error bars show standard deviation, which is less than 1% for all systems except Fast Paxos, which had higher variance.

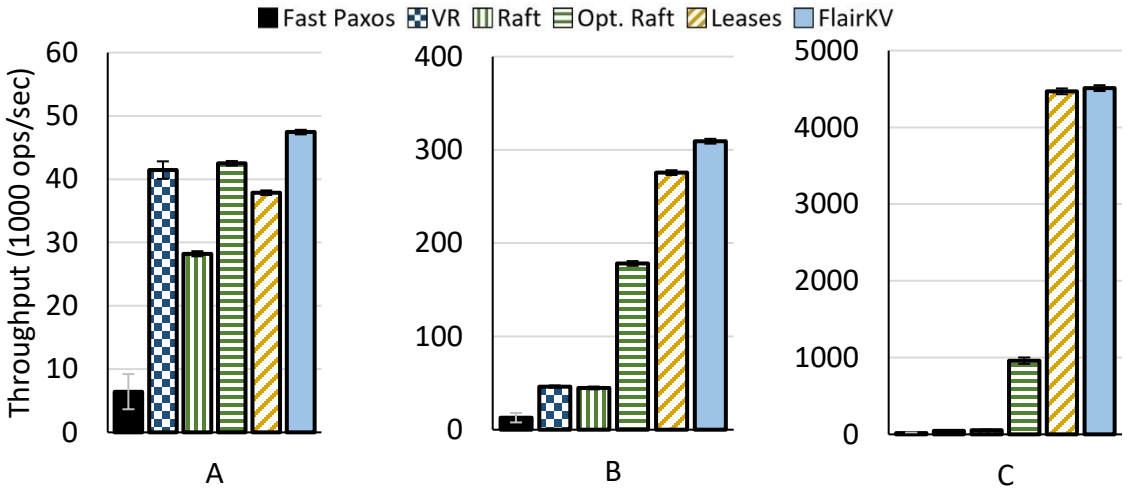


Figure 22. System's throughput (5 replicas, Zipf workload) using the YCSB workloads A, B, and C with Zipf key popularity distribution. Error bars show standard deviation which is less than 1% for all systems except Fast Paxos, which had higher variance.

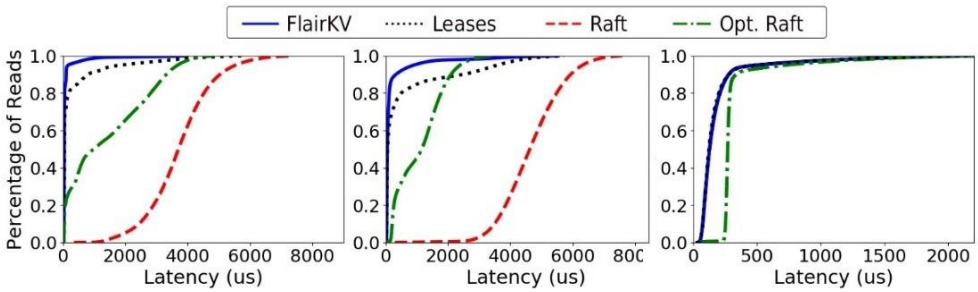


Figure 23. Latency CDF (5 replicas, uniform worload). The figures shows the latency CDF for read requests using the Uniform distribution under workload A (a), B (b), and C (c).

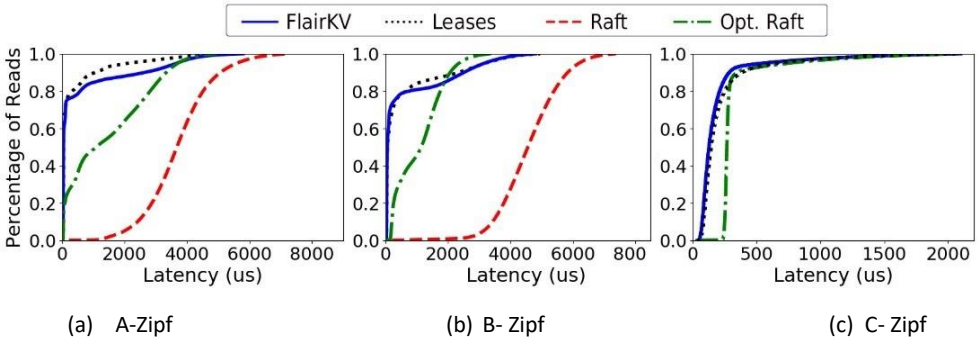


Figure 24. Latency CDF (5 replicas, Zipf worload). The figures shows the latency CDF for read requests using the Zipf distribution under workload A (a) and B (b), and C (c).

Second, when an object is not stable, Leases incurs extra latency, while FlairKV knows that the object is not stable and forwards the read requests for that object directly to the leader; Leases clients send the request to one of the followers, which redirects it to the leader. Third, Leases write operations need to reach all the followers, while FlairKV writes only need a majority. FlairKV achieves 2.4 times higher throughput than OptRaft, and at least 9 times higher than Raft, VR, and Fast Paxos.

Figure 21.A shows the throughput under write-intensive workload A. FlairKV achieves the highest performance, which is around 13% higher than Leases and OptRaft, and around 16% and 71% higher performance than VR and Raft, respectively. Fast Paxos achieves the lowest throughput. We note that the performances of Raft, VR, and Fast Paxos do not change significantly across the workloads, as reads still involve a majority of the followers.

Under the Zipf popularity distribution (Figure 22), FlairKV achieves comparable performance improvement, with a slight reduction in throughput under workloads A and B due to increased contention on the popular keys. Due to higher contention, 6.5% of the reads are redirected to the leader

in Leases, compared to only 0.3% in FlairKV, as FlairkKV can detect that there is a concurrent write to an object and directly forward the read to the leader.

We measured the operations latency under YCSB workloads A, B, and C. Figure 23 shows the latency of FlairKV, Leases, OptRaft, and Raft. Under the uniform distribution workload Figure 23 (a) and (b), FlairKV lowers the latency for the slowest 40% of operations by up to 81% relative to Leases. Figure 23 (c) shows that for a read-only workload, both FlairKV and Leases achieve similar latency, up to 52% lower than OptRaft. We excluded Raft from the figure because it had a poor performance under heavy read-only workload. Under the Zipf workload, FlairKV achieves up to 50% lower latency relative to Leases for workload B (Figure 24(b)). FlairKV and Leases achieve comparable latency under the write-heavy workload A. For workload C (Figure 24(c)), FlairKV achieved up to 28% lower latency than Leases for the slowest 20% of operations. Both FlairKV and Leases achieve more than 45% lower median latency than OptRaft.

Under all workloads, FlairKV significantly improves operation's latency relative to OptRaft and Raft. The median latency of FlairKV is 1.4% of Raft's latency and 4-6% of OptRaft's latency.

Chapter 8
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 the experience building FlairKV and presented a number of techniques to cope with the restrictions of the current programmable switches. I hope my experience informs a new generation of distributed systems that co-design network protocols and functionalities with system operations.

Bibliography

[1] 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.

[2] 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.

[3] B. Calder, J. Wang, A. Ogus *et al.*, "Windows Azure Storage: a highly available cloud storage service with strong consistency," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, 2011.

[4] J. C. Corbett, J. Dean, M. Epstein *et al.*, "Spanner: Google's globally-distributed database," in *Proceedings of the USENIX conference on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, USA, 2012.

[5] 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.

[6] H. Attiya and J. Welch, *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, Inc., 2004.

[7] L. Lamport, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18-25, 2001.

[8] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the USENIX Annual Technical Conference*, Philadelphia, PA, 2014.

[9] 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.

[10] B. Liskov and J. Cowling, "Viewstamped replication revisited," Technical Report MIT-CSAIL-TR-2012-021, MIT, 2012.

[11] 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.

[12] 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.

[13] 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.

[14] *Swift's documentation*. (April 14, 2019) https://docs.openstack.org/swift/stein/index.html.

[15] *Redis*. (April 14, 2019) https://redis.io.

[16] *Apache Cassandra*. (April 14, 2019) https://cassandra.apache.org.

[17] 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.

[18] 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.

[19] 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.

[20] 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, pp. 107-118.
[21] 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.
[22] D. Mazieres, "Paxos made practical," Technical Report on http://www.scs.stanford.edu/~dm/home/papers/paxos.pdf.
[23] *NOPaxos consensus protocol*. (April 14, 2019) https://github.com/UWSysLab/NOPaxos.
[24] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," in *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, Washington, 2006.
[25] 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.
[26] X. Jin, X. Li, H. Zhang *et al.*, "NetCache: Balancing Key-Value Stores with Fast In-Network Caching," in *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, 2017.
[27] X. Li, R. Sethi, M. Kaminsky *et al.*, "Be fast, cheap and in control with SwitchKV," in *Proceedings of the Usenix Conference on Networked Systems Design and Implementation (NSDI)*, Santa Clara, CA, 2016.
[28] *LogCabin storage system*. (April 14, 2019) https://logcabin.github.io.
[29] *P4*. (April 14, 2019) https://p4.org.
[30] *Yahoo! Cloud Serving Benchmark in C++, a C++ version of YCSB*. (April 14, 2019) https://github.com/basicthinker/YCSB-C.
[31] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133-169, 1998.
[32] *Barefoot Tofino*. (April 14, 2019) https://www.barefootnetworks.com/products/brief-tofino/.
[33] *Cavium / XPliant*. (April 14, 2019) https://origin-www.marvell.com/documents/netpxrx94dcdhk8sksbp/.
[34] 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.
[35] B. Cully, J. Wires, D. Meyer *et al.*, "Strata: High-performance scalable storage on virtualized non-volatile memory," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2014, pp. 17-31.
[36] N. Handigol, M. Flajslik, S. Seetharaman *et al.*, "Aster* x: Load-balancing as a network primitive," in *GENI Engineering Conference (Plenary)*, 2010, pp. 1-2.
[37] R. Wang, D. Butnariu, and J. Rexford, "OpenFlow-based server load balancing gone wild," in *Proceedings of the USENIX conference on Hot topics in management of internet, cloud, and enterprise networks and services*, Boston, MA, 2011.
[38] A. K. Nayak, A. Reimers, N. Feamster *et al.*, "Resonance: dynamic access control for enterprise networks," in *Proceedings of the ACM workshop on Research on enterprise networking*, Barcelona, Spain, 2009.
[39] A. J. Mashtizadeh, M. Cai, G. Tarasuk-Levin *et al.*, "XvMotion: unified virtual machine migration over long distance," in *Proceedings of the USENIX Annual Technical Conference*, Philadelphia, PA, 2014.
[40] A. Lara, A. Kolasani, and B. Ramamurthy, "Network innovation using openflow: A survey," *IEEE communications surveys & tutorials*, vol. 16, no. 1, pp. 493-512, 2014.

[41] D. R. K. Ports, J. Li, V. Liu *et al.*, "Designing distributed systems using approximate synchrony in data center networks," in *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*, Oakland, CA, 2015.
[42] J. Li, E. Michael, N. K. Sharma *et al.*, "Just say no to paxos overhead: replacing consensus with network ordering," in *Proceedings of the USENIX conference on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, USA, 2016.
[43] X. Jin, X. Li, H. Zhang *et al.*, "Netchain: scale-free sub-RTT coordination," in *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*, Renton, WA, USA, 2018.
[44] L. Lamport, D. Malkhi, and L. Zhou, "Vertical paxos and primary-backup replication," in *Proceedings of the ACM symposium on Principles of distributed computing*, Calgary, AB, Canada, 2009.
[45] H. T. Dang, D. Sciascia, M. Canini *et al.*, "NetPaxos: consensus at network speed," in *Proceedings of the ACM SIGCOMM Symposium on Software Defined Networking Research*, Santa Clara, California, 2015.
[46] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: building efficient replicated state machines for WANs," presented at the Proceedings of the 8th USENIX conference on Operating systems design and implementation, San Diego, California, 2008.
[47] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in Egalitarian parliaments," presented at the Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, Farminton, Pennsylvania, 2013.
[48] *Data Center: Load Balancing Data Center*. (April 14, 2019) https://learningnetwork.cisco.com/docs/DOC-3438.
[49] L. A. Barroso and U. Hoelzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 2009, p. 120.
[50] N. McKeown, T. Anderson, H. Balakrishnan *et al.*, "OpenFlow: enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69-74, 2008.
[51] *ettcd: Distributed reliable key-value store for the most critical data of a distributed system*. (April 14, 2019) https://github.com/etcd-io/etcd.
[52] *RethinkDB: the open-source database for the realtime web*. (April 14, 2019) https://www.rethinkdb.com/.
[53] *Open Network Operating System (ONOS) - Cluster Coordination*. https://wiki.onosproject.org/display/ONOS/Cluster+Coordination.
[54] *Apache Kudu - Fast Analytics on Fast Data*. (April 14, 2019) https://kudu.apache.org/.
[55] *Hashicorp Raft implementation*. (April 14, 2019) https://github.com/hashicorp/raft.
[56] *The Raft Consensus Algorithm*. (April 14, 2019) https://raft.github.io/.
[57] *Barefoot P4 Studio*. (April 14, 2019) https://www.barefootnetworks.com/products/brief-p4-studio/.
[58] *P4 v16 Portable Switch Architecture (PSA)*. (April 14, 2019) https://p4.org/p4-spec/docs/PSA-v1.0.0.html.
[59] L. Lamport, "Fast paxos," *Distributed Computing*, vol. 19, no. 2, pp. 79-103, 2006.
[60] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *USENIX Annual Technical Conference (USENIX ATC)*, 2014.
[61] 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.

[62] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: an engineering perspective," in *Proceedings of the ACM symposium on principles of distributed computing*, Portland, Oregon, USA, 2007.
[63] D. Mazieres, "Paxos made practical," Technical Report on http://www.scs.stanford.edu/~dm/home/papers/paxos.pdf, 2007.

Appendix A
Proof of FLAIR Safety

1 FLAIR Correctness

FLAIR uses the underlying leader-based consensus protocol to process write requests and to serve reads from the leader. FLAIR extends these protocols with the ability to serve reads from followers. The rest of this proof focuses on proving the safety of reads in FLAIR.

Consensus protocol properties. First, we state the main properties of a target leader-based consensus protocol:

Property 1. At any time, there is at most one active leader in the system, that is followed by the majority of the nodes, and is the only node that can commit new values. This leader has the highest term id in the system.

Property 2. Reads processed by the leader are always linearizable.

Property 3. If an operation at index *i* in the log is committed, then every operation with an index smaller than *i* is committed as well.

Property 4. 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.

We note that all major leader-based consensus protocols (e.g., Raft [60], Viewstamped replication [10, 61], DARE [20], Zookeeper [2], and multi-Paxos implementations [62, 63]) hold these properties.

Definitions. Before proving that FLAIR guarantees safety, we need to define a few properties.

Definition. We say the switch is *active* if it has an active leader-switch session. If the switch is not active, it will drop all FLAIR requests and replies, rendering the system unavailable. Consequently, the proof focuses on proving safety when the switch is active.

Definition. We say a kgroup is *stable* when there are no outstanding write requests in the system that *may modify* the objects in the kgroup.

Definition. We say a request or a reply is *valid* when the session id in the request match the leader information and in the reply match the switch information. Invalid requests/replies are dropped. Replicas use the information in the request to fill the fields of the reply. Consequently, request-reply pairs that span multiple sessions are dropped.

The proof focuses on proving safety with valid requests/replies.

Assumptions. FLAIR assumes the following environment properties (Note that the underlying consensus protocol may have more stringent assumptions):

- The network is unreliable and asynchronous, as there are no guarantees that packets will be received in a timely manner or delivered at all.
- There is no bound on the time a node or the switch takes to process a packet.
- Clocks are not synchronized, and there is no bound on the clock's drift rate.
- Nodes fail following the fail-stop model in which nodes may stop working but will never send erroneous messages (i.e., no byzantine failures).

Moreover, we need to define a few terms. In the following all times are relative to the switch's clock:

- $time(w)$ is the time a request/reply w was processed by the switch.
- $seq(w)$ is the sequence number of a request w.
- $wlseq_{switch}(t)$ is the largest sequence number issued by the switch of all write requests that have been received and processed by the switch before or at time t, i.e., $wlseq_{switch}(t)$ is the sequence number of a write request $w_j \mid seq(w_j) > seq(w_i) \; \forall \; w_i \neq w_j$ and $time(w_i) \leq t$.
- $rlseq_{switch}(t)$ is the largest sequence number of all write replies that have been processed by the switch before or at time t, i.e., $rlseq_{switch}(t)$ is the sequence number of a write reply $r_j \mid seq(r_j) > seq(r_i) \; \forall \; r_i \neq r_j$ and $time(r_i) \leq t$.
- $lseq_{leader}(t)$ is the largest sequence number of all write requests that have been processed by the leader before or at time t (time relative to the switch's clock).

Now, we have all the definitions to present the proof.

We will prove the safety property for the simple case in which there is a single kgroup in the system, and the kgroup has a single object (*obj*). All read and write requests access this single object. At the end of the proof, we will generalize it to multiple kgroups with multiple objects.

When the switch is active, the kgroup can be in one of two states: unstable or stable. Requests to unstable kgroups are forwarded to the leader and therefore are linearizable (Property 2). For stable kgroup, the switch forwards the read request to one of the followers included in the

consistent_followers bitmap. To prove FLAIR safety for the stable kgroup we need to prove that while a kgroup is stable, the value of the kgroup object does not change and the followers included in the consistent_followers bitmap in the kgroup entry hold the latest committed value of the object (Section 1.2), then we need to prove that read requests processed by the followers are safe (Section 1.3).

1.1 Session Start Process

The safety of FLAIR relies on the following theorem.

Theorem 1. At any moment in time there is at most one switch that is accessible by the FLAIR packets and has an active session.

Proof. Proof by contradiction. Let's assume at time t_o is the first moment in which the system had two active and accessible switches s_a, and s_b. Without loss of generality, let's assume s_b is the later switch to be activated by the leader. This means the leader has just started a new session with s_b. But before starting a session the leader asks the central controller to neutralize the old session, meaning reroute all FLAIR traffic from the old switch to the new one. Consequently, it is not possible for s_a to be accessible by FLAIR packets. □

1.2 Kgroup Stability

Lemma 1. At any moment in time t_o the following inequality holds: $wlseq_{switch}(t_o) \geq lseq_{leader}(t_o) \geq rlseq_{switch}(t_o)$

Proof. The switch sequentially processes all write requests and assigns a unique and strictly increasing sequence number for every write request. For the left side of the inequality, the switch always processes a write request before sending it to the leader. Hence, at all times, $wlseq_{switch}(t_o) \geq lseq_{leader}$. For the right side of the inequality, the leader will receive the write request and processes it before sending a reply. A write reply has the same sequence number as the corresponding write request. Hence, at all times, $lseq_{leader}(t_o) \geq rlseq_{switch}(t_o)$. □

Lemma 1 implies the following corollary:

Corollary 1.

If at time t_o $wlseq_{switch}(t_o) = rlseq_{switch}(t_o)$, then $wlseq_{switch}(t_o) = lseq_{leader}(t_o) = rlseq_{switch}(t_o)$. □

Lemma 2. At any moment in time t_o, if a request w_l has a sequence number $seq(w_l) = wlseq_{switch}(t_o)$, then w_l is the last write request processed by the switch up to time t_o, and sequence number in the kgroup entry seq_num = $seq(w_l)$.

Proof. The switch processes all packets sequentially in a pipeline. On every write, the switch atomically increments the session_seq_num in the session array, marks the kgroup entry unstable, and updates the seq_num in the kgroup entry. The fact that request w_l has the largest sequence number signifies that it was the last request to be processed by the switch up to time t_o. □

Lemma 3. If at time t_o $wlseq_{switch}(t_o) = rlseq_{switch}(t_o)$, then the last committed value of *obj* at time t_o is the value written by the request w_l with $seq(w_l) = wlseq_{switch}(t_o)$.

Proof. The fact that $wlseq_{switch}(t_o) = rlseq_{switch}(t_o)$ implies that $seq(w_l) = wlseq_{switch}(t_o) = lseq_{leader}(t_o)$ (Corollary 1), meaning that w_l is the last write request that has been processed by the leader up to time t_o.

At all times, the leader keeps track of the largest sequence number (largest_seq_num = $lseq_{leader}(t)$) processed in the current session. The leader drops every write request with a sequence number smaller than largest_seq_num. Consequently, regardless of the order in which the write requests are processed, when the leader processes w_l, it will set the largest_seq_num to equal $seq(w_l)$ and will drop any unprocessed requests with $time(w) < t_o$. Consequently, at time t_o, the last committed value is the value written by w_l. □

Now we can prove the object stability lemma.

Lemma 4. In any time interval $[t_o, t_1]$,

if $wlseq_{switch}(t_o) = rlseq_{switch}(t_o) = wlseq_{switch}(t_1)$, then the object *obj* is stable (was not modified) in the period $[t_o, t_1]$.

Proof. Assume that the write request w_l has a $seq(w_l) = wlseq_{switch}(t_1)$. The fact that $wlseq_{switch}(t_o) = rlseq_{switch}(t_o)$ implies that w_l is last request that has been processed by the leader up to time t_o (Lemma 3).

The fact that $seq(w_l) = wlseq_{switch}(t_o) = wlseq_{switch}(t_1)$ signifies that no new write requests have been processed by the switch in the interval $[t_o, t_1]$, and w_l is still the last request that has been processed by the leader up to time t_1. Consequently, the value of the object did not change in the interval $[t_o, t_1]$. □

Now we prove the stability of the kgroup data.

Lemma 5. If at time t_o $wlseq_{switch}(t_o) = rlseq_{switch}(t_o)$, then the kgroup is stable and the fields of the kgroup entry (consistent_followers bitmap and log_idx) have values equal to the fields of the write reply r_l with $seq(r_l) = wlseq_{switch}(t_o)$.

Proof. Assume that a write request w_l has $seq(w_l) = wlseq_{switch}(t_o)$ with a corresponding reply r_l with $seq(r_l) = seq(w_l)$, then seq_num in the kgroup entry at time t_o equals $seq(w_l)$ (Lemma 2).

The only time the switch will mark a kgroup as stable and update the consistent_followers bitmap and log_idx fields is when the switch receives a write reply with a sequence number equal to the seq_num in the kgroup entry.

For all other write replies with $seq(r) \neq$ seq_num, the switch will forward them to the client without updating the kgroup entry. Consequently, at time t_o, the last values of the consistent_followers and log_idx fields in the kgroup entry are the value written by r_l. □

Lemma 6. In any time interval $[t_o, t_1]$,

if $wlseq_{switch}(t_o) = rlseq_{switch}(t_o) = wlseq_{switch}(t_1)$, then the kgroup is marked stable in the period $[t_o, t_1]$ and the kgroup fields (consistent_followers bitmap and log_idx) did not change in this period.

Proof. Assume that a write reply r_l has a sequence number $seq(r_l) = wlseq_{switch}(t_o) = rlseq_{switch}(t_o)$. This signifies that when the switch processed r_l at $time(r_l) \leq t_o$, it marked the kgroup stable and set the kgroup entry fields to the values in the r_l fields (Lemma 5). The fact that $seq(r_l) = wlseq_{switch}(t_1)$ signifies that r_l is still the last reply processed by the switch at time t_1 and the kgroup is still stable. □

Now we have all the facts to prove the main stability property.

Theorem 2 (Object Stability). During any period $[t_o, t_1]$ in which a kgroup is marked stable, there are no updates to the kgroup object, and the followers included in the consistent_followers field have the latest committed value for *obj*.

Proof. In any time interval $[t_o, t_1]$, the kgroup is stable iff $wlseq_{switch}(t_o) = rlseq_{switch}(t_o) = wlseq_{switch}(t_1)$ (Lemma 6), the object *obj* is stable in the period $[t_o, t_1]$ (Lemma 4), and the value of the consistent_followers bitmap is stable, and has the value of a write r_l with $seq(r_l) = wlseq_{switch}(t_1)$ (Lemma 6).

A leader will include a follower in the consistent_followers only if the follower acknowledges the write operation. Following from Properties 3 and 4, those followers that acknowledged the write have an identical log to the leader's up to that log entry and hence have a consistent value for the object. □

1.3 Safety

The switch forwards a read request for a stable kgroup to one of the consistent followers. The reply from those followers is linearizable unless the object has been modified after the read request is processed by the switch and before the switch receives the follower's reply. To preserve safety, the switch performs a safety check on every read reply to detect stale replies.

Theorem 3 (Follower Read Reply Safety). The follower's read replies that the switch forwards to the client are linearizable.

Proof. The leader will send a read request to one of the followers in the consistent_followers bitmap only if a kgroup is stable. Those followers have a consistent version of the object that is identical to the leader's version (Theorem 1).

The switch sets the sequence number of a read request to match the sequence number of the kgroup entry. Also, the follower sets the SEQ sequence number of read replies to equal the SEQ sequence number of the corresponding read request.

The switch will only forward a read reply to a client from a follower if the reply passes the following safety check: The kgroup is stable and the sequence number of the reply matches the sequence number of the kgroup. This indicates that no writes occurred since the read was processed by the switch and the object is still consistent at the follower. □

Now we have all the facts to prove the main safety theorem.

Theorem 4 (Read Safety). FLAIR guarantees linearizability of client reads at all times.

Proof. The switch will only process requests when it is active. When the switch is active, a kgroup can be in one of two states: unstable or stable. When a kgroup is not stable, reads are linearizable as they are processed by the leader (Property 2). When a kgroup is stable, the switch will forward requests to one of the followers included in the consistent_followers field. When the switch receives the read reply, if the reply passes the safety check, it is forwarded to the client and is linearizable (Theorem 2). If a read reply does not pass the safety check, the switch will drop the reply and resubmit the read

request to the leader. Leader read replies are always linearizable (Property 1). Consequently, FLAIR reads are always linearizable. □

Generalization.

Multi-kgroup support. FLAIR does not support multi-object transactions. FLAIR guarantees linearizability only per object and does not guarantee linearizability of operations spanning multiple objects.

Generalization to multiple objects per kgroup. The switch treats all the objects in a kgroup as a single object. If the switch receives a write operation to any object in a kgroup, the kgroup entry is marked unstable. The kgroup is marked stable only when the last write to the kgroup is acknowledged by the leader. Consequently, having multiple objects per kgroup can only affect performance as it can lead to marking an object unstable and forwarding its reads to the leader only because another object in the kgroup is being updated. These false positives do not affect safety. □

Appendix B
Formal specification for FLAIR

MODULE *FLAIR*

Formal specification for *FLAIR*.

EXTENDS *Naturals*, *FiniteSets*, *Sequences*, *TLC*, *Reals*

Constants

The set of replicas
CONSTANTS *Replicas*

Replicas states.
CONSTANTS *Follower*, *Leader*

Replicas running state
CONSTANTS *ReplicaUpState*, *ReplicaDownState*

CONSTANTS *SwitchIp*, Ip address of the switch
SwitchKGroupsNum, Number of key groups
SwitchStateActive,
SwitchStateInactive

Message Types
CONSTANTS *ClientReadRequest*, Read request type
ClientWriteRequest, Write request type
WriteResponse, Write response type
ReadResponse, read response type
Read request from the switch to a replica
InternalReadRequest,
Write request from the switch to a replica
InternalWriteRequest,
A request from the leader to the replicas
to append a *log* entry
AppendEntriesRequest,
An append entry resposne from a replica
to the leader
AppendEntriesResponse

The set of possible keys and values in a client request
CONSTANTS *ValueSpace*, *KeySpace*

Special reserved value
CONSTANTS *Nil*

Variables

Replica *vars*

VARIABLE *state*, The replica's state (Follower, or *Leader*).
log, Log of all committed and uncommitted operations
commitIndex, The index of the highest committed index in the *log*
currentTerm, Current term number
isActive, Is the replica up or down
replicaSession The latest switch id seen by the leader

$replicaVars \triangleq \langle state, log, commitIndex, currentTerm, isActive, replicaSession \rangle$

Leader *vars*. The following variables are used only on leaders:

VARIABLE *nextIndex*, The next entry to send to each follower.
matchIndex, The entry index for which a follower *log* matches the leader *log*. This used to calculate commit index
replicaKGroups A map from a key hash to the last received sequence number for the associated *KGroup*

$leaderVars \triangleq \langle nextIndex, matchIndex, replicaKGroups \rangle$

Switch *vars*

VARIABLES *switchKGourpArray*, Array to maintain information about each *KGroup*
switchSeqNum, The last sequence number assigned by the switch
switchTermId, The latest term number seen by the switch
switchLeaderId, Current leader id as seen by the switch
session, Current switch id nique value for each session
switchState Is the switch up or down

$switchVars \triangleq \langle switchKGourpArray, switchState, switchTermId, switchLeaderId, switchSeqNum, session \rangle$

Messages variables

VARIABLE *messages*, messages among replicas
msgsClientSwitch, messages from the client to the switch
msgsReplicasSwitch messages from between replicas and the switch

$msgsVars \triangleq \langle messages, msgsClientSwitch, msgsReplicasSwitch \rangle$

Proof *vars*, don't appear in implementations

VARIABLES *responsesToClient*

Set of all variables

$vars \triangleq \langle msgsVars,\ replicaVars,\ leaderVars,\ switchVars,\ responsesToClient \rangle$

Helpers

Helper to *Append* an element to a set

$AddToSet(set,\ element) \triangleq set \cup \{element\}$

Helper to remove an element to a set

$RemoveFromSet(set,\ element) \triangleq set \setminus \{element\}$

Helper to return the minimum value from a set,
or undefined if the set is empty.

$Min(s) \triangleq \text{CHOOSE } x \in s : \forall\, y \in s : x \leq y$

Helper to return the maximum value from a set,
or undefined if the set is empty.

$Max(s) \triangleq \text{CHOOSE } x \in s : \forall\, y \in s : x \geq y$

Helper to choose a random element from a set

$ChooseRandomly(set) \triangleq \text{CHOOSE } i \in set : \text{TRUE}$

Helper to return the index of the *KGroup* given a hash

$getIndexFromHash(hash) \triangleq (hash\%SwitchKGroupsNum) + 1$

The set of all quorums. This just calculates simple majorities, but the only
important property is that every quorum overlaps with every other.

$Quorum \triangleq \{i \in \text{SUBSET } (Replicas) : Cardinality(i) * 2 > Cardinality(Replicas)\}$

Helper to return the term of the last entry in a *log*,
or 0 if the *log* is empty.

$LastTerm(xlog) \triangleq \text{IF } Len(xlog) = 0 \text{ THEN } 0 \text{ ELSE } xlog[Len(xlog)].term$

Helper to find list of replicas that agree on a *log* index

$$AgreeIndex(index,\ logs,\ leaderId) \triangleq \\ \{leaderId\} \cup \{k \in Replicas : \\ \qquad \land\ k \neq leaderId \\ \qquad \land\ Len(logs[k]) \geq index \\ \qquad \land\ logs[k][index] = logs[leaderId][index]\}$$

Helper to return all *log* entries for specific key

$entriesForKey(s,\ key) \triangleq \{j \in \text{DOMAIN } log[s] : \land\ log[s][j].key = key\}$

return the max value of a set or *Nil* if the set is empty

$$indexOfLastEntry(entries) \triangleq \text{IF } Cardinality(entries) > 0 \\ \text{THEN } Max(entries) \text{ ELSE } Nil$$

helper function to get the *ID* of the leader
if there is a leader that is up, this function returns its *ID*
if there is no currect leader, this function finds the next leader based
on raft criteria and returns its *ID*

$$
\begin{array}{l}
getLeaderId \triangleq \\
\quad \text{LET } leadersList \triangleq \{s \in \text{DOMAIN } state : state[s] = \\
\qquad\qquad\qquad\qquad Leader \wedge isActive[s] = ReplicaUpState\} \\
\qquad runningReplicas \triangleq \{s \in Replicas : isActive[s] = ReplicaUpState\} \\
\qquad replicasWithNonEmptyLog \triangleq \{s \in runningReplicas : Len(log[s]) > 0\} \\
\qquad latestTerm \triangleq Max(\{log[s][Len(log[s])].term : \\
\qquad\qquad\qquad\qquad\qquad\qquad s \in replicasWithNonEmptyLog\}) \\
\qquad replicasWithLatestTerm \triangleq \{s \in replicasWithNonEmptyLog : \\
\qquad\qquad\qquad\qquad\qquad\qquad log[s][Len(log[s])].term = latestTerm\} \\
\qquad lengthOfLargestLog \triangleq Max(\{Len(log[s]) : s \in replicasWithLatestTerm\}) \\
\qquad replicasWithLongestLog \triangleq \{s \in replicasWithLatestTerm : \\
\qquad\qquad\qquad\qquad\qquad\qquad Len(log[s]) = lengthOfLargestLog\} \\
\qquad newLeaderId \triangleq \text{IF } \quad Cardinality(replicasWithLongestLog) \geq 1 \\
\qquad\qquad\qquad\qquad \text{THEN CHOOSE } i \in replicasWithLongestLog : \text{TRUE} \\
\qquad\qquad\qquad\qquad \text{ELSE IF } Cardinality(runningReplicas) > 0 \\
\qquad\qquad\qquad\qquad \text{THEN CHOOSE } i \in runningReplicas : \text{TRUE} \\
\qquad\qquad\qquad\qquad \text{ELSE } Nil \\
\quad \text{IN } \quad \text{IF } Cardinality(leadersList) > 0 \\
\qquad \text{THEN CHOOSE } s \in leadersList : \text{TRUE} \\
\qquad \text{ELSE } newLeaderId
\end{array}
$$

Helper to add a message to a set of messages.

$$Send(m) \triangleq messages' = AddToSet(messages, m)$$

Helper to remove a message from a set of messages.
Used when a replica is done processing a *replicaVarsmessage*.

$$Discard(m) \triangleq messages' = RemoveFromSet(messages, m)$$

Messages schemes and data structures

$$
\begin{array}{l}
createClinetReadRequest(key) \triangleq \\
\quad [mtype \quad\quad \mapsto ClientReadRequest, \\
\quad\ mkey \quad\quad \mapsto key, \\
\quad\ mhash \quad\quad \mapsto key]
\end{array}
$$

$$
\begin{array}{l}
createClinetWriteRequest(key, value) \triangleq \\
\quad [mtype \quad \mapsto ClientWriteRequest, \\
\quad\ mkey \quad \mapsto key, \\
\quad\ mvalue \quad \mapsto value, \\
\quad\ mhash \quad \mapsto key]
\end{array}
$$

When the switch forwards a read request, it adds:
1 − *logIndex*: The replica should *executereplicaVars* this index
before serving the request
2 − *seqNum*: Which is used to ensure the linearizability
of the request response

$createInternalReadRequest(msg,\ KGroup,\ forwardTo) \triangleq$
$[mtype \mapsto InternalReadRequest,$
$mkey \mapsto msg.mkey,$
$mhash \mapsto msg.mhash,$
$msession \mapsto session,$
$mterm \mapsto switchTermId,$
$mleaderId \mapsto switchLeaderId,$
$mlogIndex \mapsto KGroup.logIndex,$
$mkGroupSeqNum \mapsto KGroup.seqNum,$
$msource \mapsto SwitchIp,$
$mdest \mapsto forwardTo]$

When the switch forwards a write request, it adds a
sequence number that is used to order write requests.
The leader processes the requests in order.

$createInternalWriteRequest(msg,\ KGroup) \triangleq$
$[mtype \mapsto InternalWriteRequest,$
$mkey \mapsto msg.mkey,$
$mvalue \mapsto msg.mvalue,$
$mhash \mapsto msg.mhash,$
$msession \mapsto session,$
$mterm \mapsto switchTermId,$
$mleaderId \mapsto switchLeaderId,$
$mkGroupSeqNum \mapsto KGroup.seqNum,$
$msource \mapsto SwitchIp,$
$mdest \mapsto switchLeaderId]$

$createReadResponse(m,\ i,\ leaderId,\ value,\ status,\ logIndex) \triangleq$
$[mtype \mapsto ReadResponse,$
$mkey \mapsto m.mkey,$
$mvalue \mapsto value,$
$mhash \mapsto m.mhash,$
$mstatus \mapsto status,$
$mlogIndex \mapsto logIndex,$
$mkGroupSeqNum \mapsto m.mkGroupSeqNum,$
$mterm \mapsto currentTerm[i],$
$mleaderId \mapsto leaderId,$
$mallLogs \mapsto log,$ for correctness check only
$mcommitIndex \mapsto commitIndex,$
$msession \mapsto m.msession,$

$msource \mapsto i,$
$mdest \mapsto SwitchIp]$

$createWriteResponse(i,\ logEntry,\ index,\ status,\ replicaIds) \triangleq$
$[mtype \mapsto WriteResponse,$
$mkey \mapsto logEntry.key,$
$mvalue \mapsto logEntry.value,$
$mhash \mapsto logEntry.hash,$
$mstatus \mapsto status,$
$mlogIndex \mapsto index,$
$mkGroupSeqNum \mapsto logEntry.seqNum,$
$msession \mapsto logEntry.switchId,$
$mreplicaIds \mapsto replicaIds,$
$mterm \mapsto currentTerm[i],$
$mallLogs \mapsto log,$ for correctness check only
$mcommitIndex \mapsto commitIndex,$
$msource \mapsto i,$
$mdest \mapsto SwitchIp]$

$createKGroup(leaderAcked,\ replicasIds,\ seqNum,\ logIndex) \triangleq$
$[leaderAcked \mapsto leaderAcked,$
$replicasIds \mapsto replicasIds,$
$seqNum \mapsto seqNum,$
$logIndex \mapsto logIndex]$

$createLogEntry(i,\ msg) \triangleq$
$[term \mapsto currentTerm[i],$
$key \mapsto msg.mkey,$
$value \mapsto msg.mvalue,$
$hash \mapsto msg.mhash,$
$seqNum \mapsto msg.mkGroupSeqNum,$
$switchId \mapsto msg.msession]$

$createResHistoryEntry(msg,\ tag) \triangleq$
$[msg \mapsto msg,$
$switchKGroupEntry \mapsto switchKGourpArray[getIndexFromHash(msg.mhash)],$
$tag \mapsto tag]$

Variables initialization

$InitSwitchVars \triangleq$
Initially the switch is inactive and

```
    KGroupArray is stable
    ∧ switchKGourpArray = [i ∈ 1 .. SwitchKGroupsNum
                            ↦ createKGroup(TRUE, {}, Nil, Nil)]
    ∧ switchState       = SwitchStateInactive
    ∧ switchTermId      = 0
    ∧ switchLeaderId    = Nil
    ∧ switchSeqNum      = 0
    ∧ session           = 0

InitMsgsSets ≜
    ∧ msgsClientSwitch      = {}
    ∧ msgsReplicasSwitch    = {}
    ∧ messages              = {}


InitReplicaVars ≜
    ∧ currentTerm = [i ∈ Replicas ↦ 1]
    ∧ state       = [i ∈ Replicas ↦ Follower]
    ∧ isActive    = [i ∈ Replicas ↦ ReplicaUpState]
    ∧ replicaSession = [i ∈ Replicas ↦ 0]
    ∧ log         = [i ∈ Replicas ↦ ⟨⟩]
    ∧ commitIndex = [i ∈ Replicas ↦ 0]

InitLeaderVars ≜
    ∧ nextIndex     = [i ∈ Replicas ↦ [j ∈ Replicas ↦ 1]]
    ∧ matchIndex    = [i ∈ Replicas ↦ [j ∈ Replicas ↦ 0]]
    ∧ replicaKGroups = [i ∈ Replicas ↦
                          [j ∈ 1 .. SwitchKGroupsNum ↦ 0]]

Init ≜ ∧ InitReplicaVars
       ∧ InitLeaderVars
       ∧ InitSwitchVars
       ∧ InitMsgsSets
        Needed to prove safety
       ∧ responsesToClient = {}
```

Variables actions

Client actions

client sends a read request to read key k

```
IssueReadRequest(key) ≜
    ∧ LET request ≜ createClinetReadRequest(key)
      IN  msgsClientSwitch' = AddToSet(msgsClientSwitch, request)
    ∧ UNCHANGED ⟨messages, replicaVars, leaderVars,
                 switchVars, msgsReplicasSwitch, responsesToClient⟩
```

```
client issues a write request to update key k
IssueWriteRequest(key, value) ≜
    ∧ LET request ≜ createClinetWriteRequest(key, value)
      IN  msgsClientSwitch' = AddToSet(msgsClientSwitch, request)
    ∧ UNCHANGED ⟨messages, replicaVars, leaderVars,
                 switchVars, msgsReplicasSwitch, responsesToClient⟩
```

Switch actions

```
Switch state changes from Active to inactive
switchFails ≜
    ∧ switchState  =  SwitchStateActive
    ∧ switchState' =  SwitchStateInactive
    ∧ UNCHANGED ⟨replicaVars, leaderVars, msgsVars, responsesToClient,
                 switchSeqNum, session, switchKGourpArray,
                 switchLeaderId, switchTermId⟩

Switch handles a read request from a client.
The request has a hash that is mapped to a
KGroup. If the KGroup is stable, the request will
be forwarded to one of the replcias, otherwise it
will be forwarded to the leader.
SwitchHandleCleintRead(msg) ≜
  LET kGroup ≜ switchKGourpArray[getIndexFromHash(msg.mhash)]
      forwardTo  ≜ IF kGroup.leaderAcked ∧
                      kGroup.seqNum ≠ Nil
                   THEN ChooseRandomly({x ∈ kGroup.replicasIds :
                                        x ≠ switchLeaderId})
                   ELSE IF kGroup.leaderAcked ∧
                           kGroup.seqNum = Nil
                   THEN switchLeaderId
                   ELSE switchLeaderId
      internalMsg ≜ createInternalReadRequest(msg, kGroup, forwardTo)

  IN
    ∧ msgsReplicasSwitch' = AddToSet(msgsReplicasSwitch, internalMsg)
    ∧ UNCHANGED ⟨replicaVars, leaderVars, switchVars,
                 responsesToClient, messages, msgsClientSwitch⟩

Switch handles a write request from a client
SwitchHandleCleintWrite(msg) ≜
    Mark the KGroup associated with the key as unstable
  LET kGroup ≜ switchKGourpArray[getIndexFromHash(msg.mhash)]
```

```
        updatedKGroup  ≜ [kGroup EXCEPT !.leaderAcked = FALSE,
                                         !.seqNum = switchSeqNum + 1,
                                         !.replicasIds = {},
                                         !.logIndex = Nil]
        internalMsg    ≜ createInternalWriteRequest(msg, updatedKGroup)

    IN
        ∧ msgsReplicasSwitch' = AddToSet(msgsReplicasSwitch, internalMsg)
        ∧ switchKGourpArray' = [switchKGourpArray EXCEPT
                                ![getIndexFromHash(msg.mhash)] = updatedKGroup]
        ∧ switchSeqNum' =    switchSeqNum + 1  Increments the sequence number
        ∧ UNCHANGED ⟨replicaVars, leaderVars, responsesToClient,
                     messages, msgsClientSwitch, switchState,
                     switchTermId, switchLeaderId, session⟩


 Switch receives a read or write request from a client
SwitchReceiveFromClient ≜
    ∧ switchState = SwitchStateActive
    ∧ Cardinality(msgsClientSwitch) > 0
    ∧ LET msg  ≜ ChooseRandomly(msgsClientSwitch)
          type ≜ msg.mtype
      IN  ∨ ∧ type = ClientReadRequest
            ∧ SwitchHandleCleintRead(msg)
          ∨ ∧ type = ClientWriteRequest
            ∧ SwitchHandleCleintWrite(msg)


 Switch handles a read response
SwitchHandleReadResponse(msg) ≜
    ∧ ∨ ∧ msg.msource ≠ switchLeaderId  msg is from follower
        ∧ msg.mterm = switchTermId   msg.term = switch term
        ∧ msg.mstatus = TRUE    The operation was succeeded
        Map the key to a KGroup based on hash.
        The response will be sent to teh client if
        1 − msg.seqNum = KGroup.seqNum
        2 − KGroup is stable
        otherwise the request will be dropped
        ∧ LET KGroup ≜ switchKGourpArray[getIndexFromHash(msg.mhash)]
              isSeqOk ≜ KGroup.seqNum = msg.mkGroupSeqNum

          IN  ∨ ∧ isSeqOk
                ∧ KGroup.leaderAcked
                ∧ responsesToClient' = AddToSet(
                                       responsesToClient,
                                       createResHistoryEntry(msg, Nil))
```

$\lor \land \neg isSeqOk$
$\land$ UNCHANGED $\langle responsesToClient \rangle$
$\lor \land \neg KGroup.leaderAcked$
$\land isSeqOk$
$\land$ UNCHANGED $\langle responsesToClient \rangle$
$\lor \land msg.msource \neq switchLeaderId$
$\land msg.mstatus = \text{FALSE}$ The key does not exist
$\land$ UNCHANGED $\langle responsesToClient \rangle$
The operation succeeded and the response
is from the current leader. Just forward it to the client
$\lor \land msg.msource = switchLeaderId$
$\land msg.mterm = switchTermId$
$\land msg.mstatus = \text{TRUE}$
$\land responsesToClient' = AddToSet(responsesToClient,$
$createResHistoryEntry(msg, Nil))$
$\lor \land msg.msource = switchLeaderId$
$\land msg.mstatus = \text{FALSE}$ The key does not exist
$\land$ UNCHANGED $\langle responsesToClient \rangle$
$\land$ UNCHANGED $\langle replicaVars, leaderVars, switchVars, msgsVars \rangle$

Switch handles a write response from a client.
If the *res.seqNum* equals the *KGroup.seqNum*, the switch
marks the *KGroup* as stable and forwards the response to
the client. If the *res.seqNum* $<$ *KGroup.seqNum*, the switch
will not change the status of the *KGroup*, and will just
forward the response to the clinet
$SwitchHandleWriteResponse(msg) \triangleq$
$\land \lor \land msg.msource = switchLeaderId$ Message from the leader
$\land msg.mterm = switchTermId$ *Message.term = switch.term*
$\land msg.mstatus = \text{TRUE}$ The operation succeeded
$\land$ LET $KGroup \triangleq switchKGourpArray[getIndexFromHash(msg.mhash)]$
$isSeqOk \triangleq KGroup.seqNum = msg.mkGroupSeqNum$
$updatedKGroup \triangleq [KGroup$ EXCEPT
$!\ .leaderAcked = \text{TRUE},$
$!\ .replicasIds = msg.mreplicaIds,$
$!\ .logIndex = msg.mlogIndex]$
msg.seqNum = KGroup.seqNum
IN $\lor \land isSeqOk$
Mark the *KGroup* as stable
$\land switchKGourpArray' = [switchKGourpArray$ EXCEPT
$![getIndexFromHash(msg.mhash)] =$
$updatedKGroup]$
$\land responsesToClient' = AddToSet(responsesToClient,$
$createResHistoryEntry(msg, Nil))$

$msg.seqNum \,!= KGroup.seqNum$

$\lor \land \neg isSeqOk$
$\land responsesToClient' = AddToSet(responsesToClient,$
$createResHistoryEntry(msg,\ Nil))$
$\land$ UNCHANGED $\langle switchKGourpArray \rangle$

$\lor \land msg.msource \neq switchLeaderId$
$\land$ UNCHANGED $\langle switchKGourpArray,\ responsesToClient \rangle$

$\lor \land msg.mstatus = \text{FALSE}$
$\land$ UNCHANGED $\langle switchKGourpArray,\ responsesToClient \rangle$

$\land$ UNCHANGED $\langle replicaVars,\ leaderVars,\ msgsVars,$
$switchSeqNum,\ session,\ switchLeaderId,$
$switchTermId,\ switchState \rangle$

Switch receives a message from a replica

$SwitchReceiveFromReplica(msg) \triangleq$

msg term id is larger than switch term id
$\Rightarrow$ switch stops processing request by setting its status to inactive

$\lor \land switchState = SwitchStateActive$
$\land msg.msession = session$
$\land msg.mterm > switchTermId$
$\land switchState' = SwitchStateInactive$
$\land$ UNCHANGED $\langle replicaVars,\ leaderVars,\ msgsVars,$
$msgsClientSwitch,\ switchVars,\ responsesToClient \rangle$

msg is coming from an old leader
$\Rightarrow$ switch just ignore the message

$\lor \land switchState = SwitchStateActive$
$\land msg.msession = session$
$\land msg.mterm < switchTermId$
$\land$ UNCHANGED $\langle replicaVars,\ leaderVars,\ msgsVars,$
$responsesToClient,\ switchVars \rangle$

$msg.switchId$ does not match $switchId$
$\Rightarrow$ switch just ignore the message

$\lor \land switchState = SwitchStateActive$
$\land msg.msession \neq session$
$\land$ UNCHANGED $\langle replicaVars,\ leaderVars,\ msgsVars,$
$responsesToClient,\ switchVars \rangle$

Switch is active and the read response passes the safety check
$\Rightarrow$ switch processes the read response

$\lor \land switchState = SwitchStateActive$
$\land msg.msession = session$
$\land msg.mtype = ReadResponse$
$\land SwitchHandleReadResponse(msg)$

```
    Switch is active and the write response passes the safety check
    ⇒ switch processes the write response
  ∨ ∧ switchState = SwitchStateActive
    ∧ msg.msession = session
    ∧ msg.mtype = WriteResponse
    ∧ SwitchHandleWriteResponse(msg)
```

Replica actions

```
Replica i fails and stops processing msgs.
It loses everything but its currentTerm and log.
Stop(i) ≜
    ∧ isActive[i] = ReplicaUpState
    ∧ isActive'   = [isActive EXCEPT ![i] = ReplicaDownState]
    ∧ UNCHANGED ⟨leaderVars, msgsVars, switchVars, responsesToClient,
                 currentTerm, log,  commitIndex,
                 state, replicaSession⟩

Replica i becomes active. The replica stars as follower
Start(i) ≜
    ∧ isActive[i]     = ReplicaDownState
    ∧ isActive'       = [isActive EXCEPT ![i] = ReplicaUpState]
    ∧ state'          = [state EXCEPT ![i] = Follower]
    ∧ nextIndex'      = [nextIndex EXCEPT ![i] = [j ∈ Replicas ↦ 1]]
    ∧ matchIndex'     = [matchIndex EXCEPT ![i] = [j ∈ Replicas ↦ 0]]
    ∧ commitIndex'    = [commitIndex EXCEPT ![i] = 0]
    ∧ replicaKGroups' = [replicaKGroups EXCEPT ![i] =
                                    [j ∈ 1 .. SwitchKGroupsNum ↦ 0]]
    ∧ UNCHANGED ⟨msgsVars, switchVars, responsesToClient,
                 currentTerm, log, replicaSession⟩

Select a new leader if non of the replicas is a leader
This helper selects the new leader based on raft criteria.
That is, the node with the log with the highest term id and
longest log, if mutiple nodes have the same log id.
ElectLeader ≜
    ∧ LET runningReplicas ≜ {s ∈ Replicas : isActive[s] = ReplicaUpState}
          replicasWithNonEmptyLog ≜ {s ∈ runningReplicas : Len(log[s]) > 0}
          latestTerm ≜ Max({log[s][Len(log[s])].term :
                                  s ∈ replicasWithNonEmptyLog})
          replicasWithLatestTerm ≜ {s ∈ replicasWithNonEmptyLog :
                                    log[s][Len(log[s])].term = latestTerm}
          lengthOfLargestLog ≜ Max({Len(log[s]) :
                                    s ∈ replicasWithLatestTerm})
```

```
        replicasWithLongestLog ≜ {s ∈ replicasWithLatestTerm :
                                   Len(log[s]) = lengthOfLargestLog}
        newLeaderId ≜ IF   Cardinality(replicasWithLongestLog) ≥ 1
                      THEN CHOOSE i ∈ replicasWithLongestLog : TRUE
                      ELSE IF Cardinality(runningReplicas) > 0
                      THEN CHOOSE i ∈ runningReplicas : TRUE
                      ELSE Nil
        newTerm   ≜ IF newLeaderId ≠ Nil
                    THEN currentTerm[newLeaderId] + 1 ELSE Nil
        majority  ≜ IF newLeaderId ≠ Nil
                    THEN CHOOSE g ∈ Quorum : newLeaderId ∈ g
                    ELSE Nil
        newCurrentTerm ≜ [j ∈ Replicas ↦ IF j ∈ majority
                                          THEN newTerm
                                          ELSE currentTerm[j]]
    IN  ∧ Cardinality(runningReplicas) * 2 > Cardinality(Replicas)
        ∧ ∀ i ∈ runningReplicas : state[i] ∈ {Follower}
        ∧ state'      = [state EXCEPT ![newLeaderId] = Leader]
        ∧ nextIndex'  = [nextIndex EXCEPT ![newLeaderId] =
                        [j ∈ Replicas ↦ Len(log[newLeaderId]) + 1]]
        ∧ matchIndex' = [matchIndex EXCEPT ![newLeaderId] =
                        [j ∈ Replicas ↦ 0]]
        ∧ currentTerm' = newCurrentTerm
  ∧ UNCHANGED ⟨switchVars, msgsVars, responsesToClient,
               replicaKGroups, commitIndex, log,
               isActive, replicaSession⟩

 leader i populates the switch KGroup array
 with information about unstable KGroups
fillSwitchKGroup(i) ≜
  LET startIndex ≜ 1
      endIndex   ≜ Len(log[i])
       Scan the log and map each key in the log
       to its associated KGroup
      keysToKGroups ≜ [j ∈ startIndex .. endIndex ↦
                       getIndexFromHash(log[i][j].hash)]
       For each KGroup, find the last log entry
       that should be used to update the switch
       KGroup
      mapLogEntryToKGroupIndex ≜
        [j ∈ 1 .. SwitchKGroupsNum ↦
        IF Cardinality({k ∈ DOMAIN keysToKGroups : keysToKGroups[k] = j}) > 0
         THEN Max({k ∈ DOMAIN keysToKGroups : keysToKGroups[k] = j})
         ELSE Nil]
```

```
        A boolean array that indicates whether
        a log entry is committed or not
    leaderAckedFlag ≜ [j ∈ startIndex .. endIndex ↦
                          j ≤ commitIndex[i]]
        Get the set of replicas that
        acknowleged each log entry
    keysToReplicas ≜ [j ∈ startIndex .. endIndex ↦
                        IF leaderAckedFlag[j]
                         THEN AgreeIndex(j, log, i)
                         ELSE {i}]
IN      Update the switch KGroup Array to match the leader's
        KGroup. If the leader do not have any writes for a
        KGroup, then the KGroup is stable
  ∧ switchKGourpArray' =
      [j ∈ 1 .. SwitchKGroupsNum ↦
      IF   mapLogEntryToKGroupIndex[j] ≠ Nil
       THEN createKGroup(leaderAckedFlag[mapLogEntryToKGroupIndex[j]],
                         keysToReplicas[mapLogEntryToKGroupIndex[j]],
                         0, mapLogEntryToKGroupIndex[j])
       ELSE createKGroup(TRUE, Replicas, Nil, Nil)]


 Leader i activates the switch
LeaderActivateSwitch(i) ≜
    ∧ state[i] = Leader  Replica is the leader
    ∧ isActive[i]  = ReplicaUpState  Replica is active
    ∧ switchState = SwitchStateInactive  Switch is inactive
    ∧ replicaSession' = [replicaSession EXCEPT ![i] = session + 1]
    ∧ session' = session + 1  Update the replica and switch sessions
    ∧ switchTermId' = currentTerm[i]  Update switch term number
    ∧ switchLeaderId' = i  Update leader id of the switch
    ∧ switchSeqNum' = 0  Reset the sequence number
    ∧ fillSwitchKGroup(i)  Populate switch KGroup array
    ∧ switchState' = SwitchStateActive  activate the switch
    ∧ UNCHANGED ⟨leaderVars, msgsVars, responsesToClient,
                 state, log, commitIndex, currentTerm,
                 isActive⟩


 Any RPC with a newer term causes the recipient to advance its term first.
UpdateTerm(i, j, m) ≜
    ∧ m.mterm > currentTerm[i]
    ∧ currentTerm'   = [currentTerm EXCEPT ![i] = m.mterm]
    ∧ state'         = [state       EXCEPT ![i]  = Follower]
      messages is unchanged so m can be processed further.
```

$\land$ UNCHANGED $\langle switchVars,\ msgsVars,\ leaderVars,$
$responsesToClient,\ isActive,\ log,$
$commitIndex,\ replicaSession\rangle$

Responses with stale terms are ignored.
$DropStaleResponse(i,\ j,\ m) \triangleq$
$\land\ m.mterm < currentTerm[i]$
$\land$ UNCHANGED $\langle replicaVars,\ leaderVars,\ switchVars,$
$responsesToClient,\ msgsVars\rangle$

Replica i receives a read request (m) from a client.
$ReplicaReceiveReadRequest(m,\ i) \triangleq$

The request received by the leader
$\lor\ \land\ state[i] = Leader$
Check the message term numebr
$\land\ m.mterm = currentTerm[i]$
Get the index of the last committed entry
$\land$ LET $committedEntries \triangleq \{j \in$ DOMAIN $log[i] :\ \land\ log[i][j].key = m.mkey$
$\land\ j \leq commitIndex[i]\}$
$lastEntryIndex \triangleq$ IF $Cardinality(committedEntries) > 0$
THEN $Max(committedEntries)$ ELSE Nil
$success \triangleq$ IF $lastEntryIndex = Nil$ THEN FALSE ELSE TRUE
$value \triangleq$ IF $success$ THEN $log[i][lastEntryIndex].value$ ELSE Nil
IN $\land\ msgsReplicasSwitch' = AddToSet(msgsReplicasSwitch,$
$createReadResponse(m,\ i,\ getLeaderId,$
$value,\ success,\ lastEntryIndex))$
$\land$ UNCHANGED $\langle replicaVars,\ leaderVars,\ switchVars,$
$msgsClientSwitch,\ messages,$
$responsesToClient\rangle$

The request received by a follower and $msg.mlogIndex > 0$
$i.e$, the switch processed a write associated with the same
$KGroup$ of the key
$\lor\ \land\ state[i] = Follower$
$\land\ m.mterm = currentTerm[i]$ $msg.term =$ replica term
$\land\ m.mlogIndex \leq Len(log[i])$ The replica has the index
$\land\ m.mlogIndex > 0$ Switch $Kgroup$ entry is not empty
Get the last committed log entry for the requested key
$\land$ LET $logEntriesForKey \triangleq entriesForKey(i,\ m.mkey)$
$filteredEntries \triangleq \{j \in logEntriesForKey : j \leq m.mlogIndex\}$
$lastEntryIndex \triangleq$ IF $Cardinality(filteredEntries) > 0$
THEN $Max(filteredEntries)$ ELSE Nil
$requestedEntry \triangleq$ IF $Cardinality(filteredEntries) > 0$
THEN $log[i][lastEntryIndex]$ ELSE Nil

$isCommitted \triangleq m.mlogIndex \leq commitIndex[i]$
$success \triangleq$ IF $Cardinality(filteredEntries) > 0$
THEN $requestedEntry.key = m.mkey$ ELSE FALSE
$value \triangleq$ IF $success$ THEN $requestedEntry.value$ ELSE Nil
IN $\land msgsReplicasSwitch' = AddToSet(msgsReplicasSwitch,$
$createReadResponse(m,\ i,\ getLeaderId,$
$value,\ success,\ lastEntryIndex))$
$\land \lor \land isCommitted$
$\land$ UNCHANGED $\langle commitIndex \rangle$
$\lor \land \neg isCommitted$
$\land success$
$\land commitIndex' = [commitIndex$ EXCEPT $![i] = m.mlogIndex]$
$\land$ UNCHANGED $\langle leaderVars,\ switchVars,\ responsesToClient,$
$log,\ state,\ currentTerm,\ isActive,$
$msgsClientSwitch,\ messages,\ replicaSession \rangle$

The request received by a follower and $msg.mlogIndex = -1$.
i.e., the switch did not process any write associated
$\lor \land state[i] = Follower$
$\land m.mterm = currentTerm[i]$ $msg.term =$ replica term
$\land m.mlogIndex = Nil$ Switch *Kgroup* entry is empty
Get the last committed *log* entry for the requested key
$\land$ LET $committedEntries \triangleq \{j \in$ DOMAIN $log[i] : \land log[i][j].key = m.mkey$
$\land j \leq commitIndex[i]\}$
$lastEntryIndex \triangleq$ IF $Cardinality(committedEntries) > 0$
THEN $Max(committedEntries)$ ELSE Nil
$success \triangleq$ IF $lastEntryIndex = Nil$ THEN FALSE ELSE TRUE
$value \triangleq$ IF $success$ THEN $log[i][lastEntryIndex].value$ ELSE Nil
IN $\land msgsReplicasSwitch' = AddToSet(msgsReplicasSwitch,$
$createReadResponse(m,\ i,$
$getLeaderId,\ value,\ success,$
$lastEntryIndex))$

$\land$ UNCHANGED $\langle replicaVars,\ leaderVars,\ switchVars,$
$responsesToClient,\ msgsClientSwitch,$
$messages \rangle$

Leader i receives a write request.
$ReplicaReceiveWriteRequest(m,\ i) \triangleq$
Safety checks
$\lor \land m.mterm = currentTerm[i]$ $msg.term = replica.term$
$\land m.mleaderId = i$ The *leaderId* field in the message is the replica
$\land state[i] = Leader$ The replica is the leader
$\land m.msession = replicaSession[i]$ $msg.session = replica.session$
Get the highest sequence number received fot his *KGroup*

```
        ∧ LET latestSeenSeqNum ≜ replicaKGroups[i][getIndexFromHash(m.mhash)]
              entry ≜ createLogEntry(i, m)
              newLog ≜ Append(log[i], entry)
               msg.seqNum > KGroup.seqNum
          IN  ∨ ∧ m.mkGroupSeqNum > latestSeenSeqNum
                 Update the KGroup seqNum
                ∧ replicaKGroups' = [replicaKGroups EXCEPT ![i] =
                                       [@ EXCEPT ![getIndexFromHash(m.mhash)] =
                                       m.mkGroupSeqNum]]
                ∧ log' = [log EXCEPT ![i] = newLog]  Append to log
                ∧ UNCHANGED ⟨switchVars, msgsVars, responsesToClient,
                             nextIndex, matchIndex, commitIndex,
                             replicaSession, state, currentTerm, isActive⟩
              ∨ ∧ m.mkGroupSeqNum ≤ latestSeenSeqNum
                ∧ UNCHANGED ⟨replicaVars, leaderVars, switchVars,
                             responsesToClient, msgsVars⟩
    ∨ ∧ ∨ m.mterm      ≠ currentTerm[i]
        ∨ m.mleaderId  ≠ i
        ∨ state[i]     ≠ Leader
        ∨ m.msession ≠ replicaSession[i]
      ∧ UNCHANGED ⟨replicaVars, leaderVars, switchVars,
                   responsesToClient, msgsVars⟩
```

Leader i sends follower j an *AppendEntries* request containing up to 1 entry. While implementations may want to send more than 1 at a time, this spec uses just 1 because it minimizes atomic regions without loss of generality.

```
AppendEntries(i, j) ≜
    ∧ i ≠ j  Avoid sending to itself
    ∧ state[i] = Leader  The sender is the leader
    ∧ isActive[i] = ReplicaUpState  The sender is active
     Get the index of the next entry that must
     be sent to the follower j
    ∧ LET prevLogIndex ≜ nextIndex[i][j] − 1
          prevLogTerm ≜ IF prevLogIndex > 0 THEN
                           log[i][prevLogIndex].term
                        ELSE
                           0
       Send up to 1 entry, constrained by the end of the log.
      lastEntry ≜ Min({Len(log[i]), nextIndex[i][j]})
      entries ≜ SubSeq(log[i], nextIndex[i][j], lastEntry)
      m ≜ [mtype          ↦ AppendEntriesRequest,
           mterm          ↦ currentTerm[i],
           mprevLogIndex  ↦ prevLogIndex,
           mprevLogTerm   ↦ prevLogTerm,
```

```
                    mentries       ↦ entries,
                    mlog is used as a history variable for the proof.
                    It would not exist in a real implementation.
                    mlog           ↦ log[i],
                    mcommitIndex   ↦ Min({commitIndex[i], lastEntry}),
                    msource        ↦ i,
                    mdest          ↦ j]
  IN    ∧ Len(entries) > 0
        ∧ Send(m)
∧ UNCHANGED ⟨replicaVars, leaderVars, switchVars,
             msgsClientSwitch, msgsReplicasSwitch,
             responsesToClient⟩
```

Leader i advances its *commitIndex*.
This is done as a separate step from handling *AppendEntries* responses,
in part to minimize atomic regions, and in part so that leaders of
single-server clusters are able to mark entries committed.

```
AdvanceCommitIndex(i) ≜
  ∧ state[i] = Leader
  ∧ isActive[i] = ReplicaUpState
  ∧ LET The set of replicas that agree up through an index.
        Agree(index) ≜ {i} ∪ {k ∈ Replicas :
                               matchIndex[i][k] ≥ index}
        The maximum indexes for which a quorum agrees
        agreeIndexes ≜ {index ∈ 1 .. Len(log[i]) :
                         Agree(index) ∈ Quorum}
        Entries that are replicated to majorities
        but not yet committed
        IndicesToCommit ≜ {index ∈ agreeIndexes : index > commitIndex[i]}
        majoritiesPerIndex ≜ [index ∈ agreeIndexes ↦ Agree(index)]
        New value for commitIndex'[i]
        newCommitIndex ≜
          IF ∧ agreeIndexes ≠ {}
             ∧ log[i][Max(agreeIndexes)].term = currentTerm[i]
           THEN Max(agreeIndexes)
           ELSE commitIndex[i]
    IN  ∧ newCommitIndex > commitIndex[i]
        For each entry that will be committed, send a write
        response to clients
        ∧ LET indices ≜ commitIndex[i] + 1 .. newCommitIndex
              msgs ≜ [x ∈ indices ↦
                        createWriteResponse(i, log[i][x],
```

```
                                                    x, TRUE, majoritiesPerIndex[x])]
            msgsAsSet ≜ {createWriteResponse(i, log[i][x],
                                                    x, TRUE, majoritiesPerIndex[x])
                                                      : x ∈ indices}

    IN   ∧ msgsReplicasSwitch' = msgsReplicasSwitch ∪ msgsAsSet
         Update the commit index at the replica
         ∧ commitIndex' = [commitIndex EXCEPT ![i] = newCommitIndex]
         ∧ UNCHANGED ⟨leaderVars, switchVars, responsesToClient,
                      messages, msgsClientSwitch, log, currentTerm,
                      isActive, replicaSession, state⟩

Replica i receives an AppendEntries request from Replica j.
This just handles m.entries of length 0 or 1, but
implementations could safely accept more by treating
them the same as multiple independent requests of 1 entry.
HandleAppendEntriesRequest(i, j, m) ≜
  LET logOk ≜ ∨ m.mprevLogIndex = 0
              ∨ ∧ m.mprevLogIndex > 0
                ∧ m.mprevLogIndex ≤ Len(log[i])
                ∧ m.mprevLogTerm = log[i][m.mprevLogIndex].term
  IN  ∧ m.mterm ≤ currentTerm[i]
      ∧ ∨ ∧ reject request
          ∨ m.mterm < currentTerm[i]
          ∨ ∧ m.mterm = currentTerm[i]
            ∧ state[i] = Follower
            ∧ ¬logOk
        ∧ Send([mtype      ↦ AppendEntriesResponse,
                mterm      ↦ currentTerm[i],
                msuccess   ↦ FALSE,
                mmatchIndex ↦ 0,
                msource    ↦ i,
                mdest      ↦ j])
        ∧ UNCHANGED ⟨replicaVars⟩
        process the request
      ∨ ∧ m.mterm = currentTerm[i]
        ∧ state[i] = Follower
        ∧ logOk
        ∧ LET index ≜ m.mprevLogIndex + 1
          IN  ∨ already done with request
                ∧ ∨ m.mentries = ⟨⟩
                  ∨ ∧ Len(log[i]) ≥ index
                    ∧ m.mentries ≠ ⟨⟩
                    ∧ log[i][index].term = m.mentries[1].term
                This could make our commitIndex decrease (for
```

example if we process an old, duplicated request),
but that doesn't really affect anything.

$\land\ commitIndex' = [commitIndex \text{ EXCEPT } ![i] = m.mcommitIndex]$
$\land\ Send([mtype \mapsto AppendEntriesResponse,$
$\quad mterm \mapsto currentTerm[i],$
$\quad msuccess \mapsto \text{TRUE},$
$\quad mmatchIndex \mapsto m.mprevLogIndex + Len(m.mentries),$
$\quad msource \mapsto i,$
$\quad mdest \mapsto j])$
$\land\ \text{UNCHANGED } \langle log \rangle$

$\lor$ conflict: remove 1 entry
$\land\ m.mentries \neq \langle\rangle$
$\land\ Len(log[i]) \geq index$
$\land\ log[i][index].term \neq m.mentries[1].term$
$\land\ \text{LET } new \triangleq [index2 \in 1\,..\,(Len(log[i]) - 1) \mapsto log[i][index2]]$
$\quad \text{IN } log' = [log \text{ EXCEPT } ![i] = new]$

$\land\ Send([mtype \mapsto AppendEntriesResponse,$
$\quad mterm \mapsto currentTerm[i],$
$\quad msuccess \mapsto \text{FALSE},$
$\quad mmatchIndex \mapsto 0,$
$\quad msource \mapsto i,$
$\quad mdest \mapsto j])$
$\land\ \text{UNCHANGED } \langle commitIndex \rangle$

$\lor$ no conflict: append entry
$\land\ m.mentries \neq \langle\rangle$
$\land\ Len(log[i]) = m.mprevLogIndex$
$\land\ log' = [log \text{ EXCEPT } ![i] = Append(log[i],\ m.mentries[1])]$
$\land\ Send([mtype \mapsto AppendEntriesResponse,$
$\quad mterm \mapsto currentTerm[i],$
$\quad msuccess \mapsto \text{TRUE},$
$\quad mmatchIndex \mapsto m.mprevLogIndex + Len(m.mentries),$
$\quad msource \mapsto i,$
$\quad mdest \mapsto j])$
$\land\ \text{UNCHANGED } \langle commitIndex \rangle$

$\land\ \text{UNCHANGED } \langle leaderVars,\ switchVars,\ responsesToClient,$
$msgsClientSwitch,\ msgsReplicasSwitch,$
$isActive,\ currentTerm,\ state,\ replicaSession \rangle$

Replica i receives an *AppendEntries* response from Replica j

```
HandleAppendEntriesResponse(i, j, m) ≜
    ∧ m.mterm = currentTerm[i]
    ∧ ∨ ∧ m.msuccess  successful
        ∧ nextIndex'  = [nextIndex  EXCEPT ![i][j]  = m.mmatchIndex + 1]
        ∧ matchIndex' = [matchIndex EXCEPT ![i][j] = m.mmatchIndex]
      ∨ ∧ ¬m.msuccess  not successful
        ∧ nextIndex' = [nextIndex EXCEPT ![i][j] =
                            Max({nextIndex[i][j] − 1, 1})]
        ∧ UNCHANGED ⟨matchIndex⟩
    ∧ Discard(m)
    ∧ UNCHANGED ⟨replicaVars, switchVars, responsesToClient,
                 msgsClientSwitch, msgsReplicasSwitch, replicaKGroups⟩


 process a message. The message will be processed
 by its recipient based on its type
Receive(m) ≜
    LET i ≜ m.mdest
        j ≜ m.msource
    IN   Any RPC with a newer term causes the recipient to advance
         its term first. Responses with stale terms are ignored.
    ∧   isActive[i] = ReplicaUpState
    ∧   ∨ ∧ m.mtype ∈ {AppendEntriesRequest, AppendEntriesResponse}
          ∧ UpdateTerm(i, j, m)
         Append entry request from replica j to replica i
        ∨ ∧ m.mtype = AppendEntriesRequest
          ∧ HandleAppendEntriesRequest(i, j, m)
         Append entry response from replica j to replica i
        ∨ ∧ m.mtype = AppendEntriesResponse
          ∧ ∨ DropStaleResponse(i, j, m)
            ∨ HandleAppendEntriesResponse(i, j, m)

         Read request from the switch to replica i
        ∨ ∧ m.mtype = InternalReadRequest
          ∧ ReplicaReceiveReadRequest(m, i)
         Write request from the switch to replica i
        ∨ ∧ m.mtype = InternalWriteRequest
          ∧ ReplicaReceiveWriteRequest(m, i)

ReplicasReceiveRaftInternalMsgs ≜
    ∧ ∃ m ∈ messages : Receive(m)

ReplicasReceiveFromSwitch ≜
    ∧ ∃ m ∈ msgsReplicasSwitch :
            ∧ m.mdest ≠ SwitchIp ∧ Receive(m)
```

Defines how all *variables*(*Replicas*, *Client*, *Switch*)
may transition.

$$
\begin{array}{l}
Next \triangleq \\
\quad \text{client transitions} \\
\quad \land \lor \exists\, key \in KeySpace : IssueReadRequest(key) \\
\quad\quad \lor \exists\, key \in KeySpace : \exists\, value \in ValueSpace : \\
\quad\quad\quad\quad\quad\quad IssueWriteRequest(key,\, value) \\
\\
\quad \text{Switch transitions} \\
\quad\quad \lor\, switchFails \\
\quad\quad \lor\, SwitchReceiveFromClient \\
\quad\quad \lor\ \land\ Cardinality(msgsReplicasSwitch) > 0 \\
\quad\quad\quad \land \text{LET}\ messagesToSwitch \triangleq \{x \in msgsReplicasSwitch : x.mdest = SwitchIp\} \\
\quad\quad\quad\quad \text{IN}\quad \land \exists\, msg \in messagesToSwitch : SwitchReceiveFromReplica(msg) \\
\\
\quad \text{Replica transitions} \\
\quad\quad \lor \exists\, i \in Replicas : Stop(i) \\
\quad\quad \lor \exists\, i \in Replicas : Start(i) \\
\quad\quad \lor\, ElectLeader \\
\quad\quad \lor\, ReplicasReceiveRaftInternalMsgs \\
\quad\quad \lor\, ReplicasReceiveFromSwitch \\
\\
\quad \text{Leader transitions} \\
\quad\quad \lor \exists\, i \in Replicas : LeaderActivateSwitch(i) \\
\quad\quad \lor \exists\, i, j \in Replicas : AppendEntries(i, j) \\
\quad\quad \lor \exists\, i \in Replicas\quad : AdvanceCommitIndex(i)
\end{array}
$$

Safety Invariants

In the following statements, each response has the
logs of all replicas at the time it was served
by a replica (This is only for safety check and
should not be the case for real implementations)

Invariant that defines that read responses that
passes the switch to the client are linearizable.
Check the logs of all replica to get the last committed
log index that update the requested key. The returned value
"*msg.mvalue*" should equal the value in that *log* index.

$$
\begin{array}{l}
isForwardedToClientReadSafe(response) \triangleq \\
\quad \text{LET}\ msg \triangleq response.msg \\
\quad\quad\quad logEntriesForKey \triangleq entriesForKey(msg.mleaderId,\, msg.mkey)
\end{array}
$$

```
        committedEntries ≜ {j ∈ logEntriesForKey :
                            AgreeIndex(j, msg.mallLogs, msg.mleaderId)
                             ∈ Quorum ∧ j ≤ msg.mlogIndex}
        lastCommittedIndex ≜ indexOfLastEntry(committedEntries)
        lastEntry ≜ log[msg.mleaderId][lastCommittedIndex]
    IN
        ∨ ∧ msg.mstatus = TRUE
          ∧ lastCommittedIndex = msg.mlogIndex
          ∧ lastEntry.key      = msg.mkey
          ∧ lastEntry.value    = msg.mvalue

 Invariant that defines that write responses that
 passes the switch to the client are committed
 on majority of the replicas.
isForwardedToClientWriteCorrect(response) ≜
    LET msg ≜ response.msg
        isOnMajority ≜ AgreeIndex(msg.mlogIndex,
                                  msg.mallLogs, msg.msource) ∈ Quorum
    IN  isOnMajority

 Invariant that defines the correctness
 of all responses forwarded to the client
InvResponsesToClientCorrectness ≜
    ∀ res ∈ responsesToClient :
      ∨ ∧ res.msg.mtype = ReadResponse
        ∧ isForwardedToClientReadSafe(res)
      ∨ ∧ res.msg.mtype = WriteResponse
        ∧ isForwardedToClientWriteCorrect(res)

 Invariant that defines that no two entries have
 the same sequence number or log index.
InvSwitchRegisterCorrectness ≜
    ∧ ∀ i, j ∈ 1 .. SwitchKGroupsNum :
        ∨ ∧ i ≠ j
          ∧ switchKGourpArray[i].leaderAcked = TRUE
          ∧ switchKGourpArray[j].leaderAcked = TRUE
          ∧ switchKGourpArray[i].seqNum ≠ switchKGourpArray[j].seqNum
          ∧ switchKGourpArray[i].logIndex ≠ switchKGourpArray[j].logIndex
        ∨ i = j
        ∨ switchKGourpArray[i].leaderAcked = FALSE
        ∨ switchKGourpArray[j].leaderAcked = FALSE


 invariant that defines that no two leaders exist
 at the same time
InvLeaderElectionSafety ≜
```

$$
\begin{array}{ll}
\text{LET} & runningReplicas \triangleq \{i \in Replicas : isActive[i] = ReplicaUpState\} \\
& leaders \triangleq \{i \in runningReplicas : state[i] = Leader\} \\
\text{IN} & Cardinality(leaders) \leq 1
\end{array}
$$