Accelerating Reads with Consistency-Aware Network Routing

Technical Report

This technical report includes the following additional material compared to our SOSP #11 submission: a detailed proof of FLAIR safety, additional evaluation results with five-replica deployment, and the TLA+ specification.

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 [1], Viewstamped replication [2, 3], DARE [4], Zookeeper [5], and multi-Paxos implementations [6, 7]) 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, our 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.

Our 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 our 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 our main 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_indx) 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_indx 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_indx 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_indx) 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_1 with $seq(r_1) = wlseq_{switch}(t_1)$ (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 □

2 Additional Evaluation

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

In summary, the 5-replica results corroborate our findings in the paper.

2.1 Throughput Evaluation

Figure 1 shows the throughput of the six systems for workloads A, B, and C. For workload C (Figure 1.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

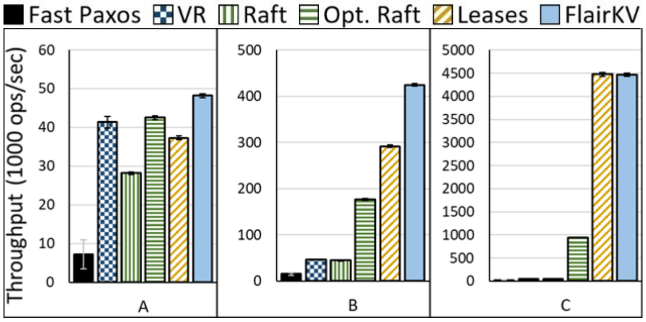
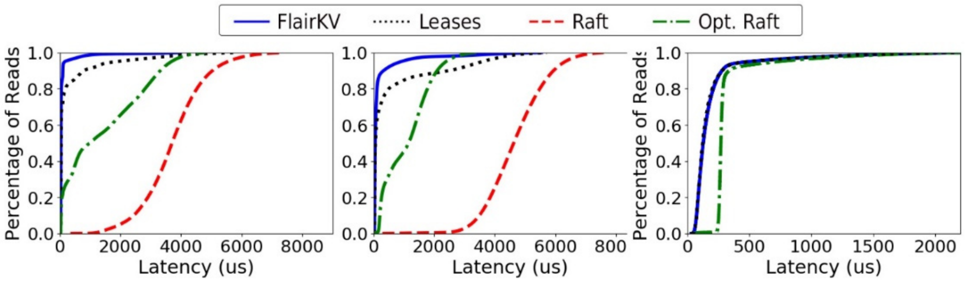
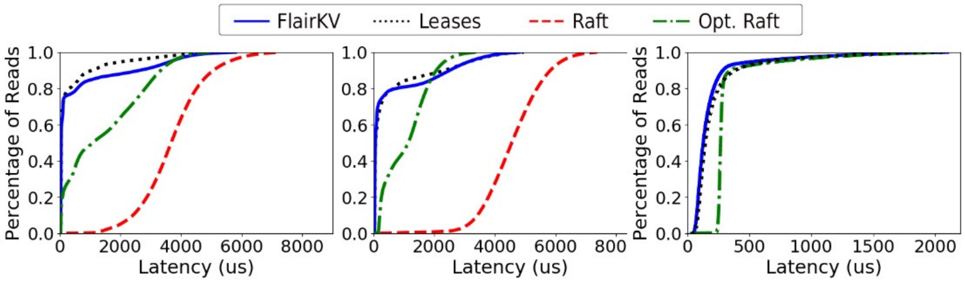


Figure 1. System's throughput using the YCSB workloads A, B, and C with uniform key popularity distribution. Error bars show standard deviation, which is less than 1% for all systems except Fast Paxos, which had higher variance.



(a) A-Uniform (b) B-Uniform (c) C-Uniform

Figure 3. Latency CDF. The figures shows the latency CDF for read requests using the Uniform distribution under workload A (a), B (b), and C (c).



(a) A-Zipf (b) B- Zipf (c) C- Zipf

Figure 4. Latency CDF. The figures shows the latency CDF for read requests using the Zipf distribution under workload A (a) and B (b), and C (c).

Fast Paxos, as these protocols contact the majority of followers for every read operation.

Figure 1.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.

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.

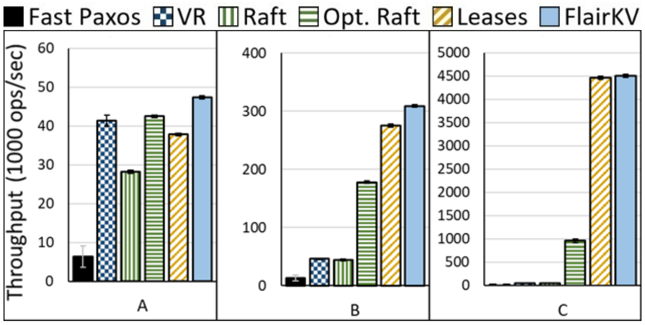


Figure 2. System's throughput using the YCSB workloads A, B, and C with Zipf key popularity distribution. Error bars show standard deviation which is less than 1% for all systems except Fast Paxos, which had higher variance.

Figure 1.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 2), 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.

2.2 Latency Evaluation

We measured the operations latency under YCSB workloads A, B, and C. Figure 3 shows the latency of FlairKV, Leases, OptRaft, and Raft. Under the uniform distribution workload Figure 3 (a) and (b), FlairKV lowers the latency for the slowest 40% of operations by up to 81% relative to Leases. Figure 3 (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 4(b)). FlairKV and Leases achieve comparable latency under the write-heavy workload A. For workload C

(Figure 4(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.

References

[1] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *USENIX Annual Technical Conference (USENIX ATC)*, 2014.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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