Scalable Reads on Skyros

Ramya Bygari
UIUC

1 Abstract

Improving the performance of storage systems while maintaining strong consistency and linearizability is a challenging problem, and recent research has been successful in this regard. Notably, several works such as CAD[1], Skyros [2], Quorum reads in Paxos[3], and SpecPaxos [4] have received wide appreciation for their ability to provide efficient storage operations with one round trip time while still ensuring durability.

Many of the existing systems are suited for a single-region deployment setting and provide high latency in geo-replicated settings. We propose Syros, which exploits the fact that if a quorum of replicas are present closer to the client, then it is possible to achieve significantly high read throughput while not compromising any other properties. We show that Syros is able to perform 50% better than existing replication protocols on a mixed read write workload and almost 300 times better for a read only workload.

2 Introduction

Many commercially deployed systems, like Cassandra [5], DynamoDB [6] etc, sacrifice strong consistency to achieve high availability. On the other hand, strongly consistent systems typically tend to use techniques like consensus to achieve strong consistency and linearizability. These properties generally come with a performance cost.

Paxos[7] is a widely used and reliable protocol used for achieving distributed consensus in large-scale distributed systems. Other variants of Paxos like Raft [8] and Viewstamped Replication [9] have also been implemented in real-world systems like Zookeeper [10]. Despite their popularity, these protocols require two round-trip times to complete a full round of read or write. In leader-based protocols like Paxos, the leader orders and replicates the client's request on a quorum of followers for durability, and then executes the requests before sending an acknowledgement to the client. Hence, all types of requests in these protocols take 2RTTs to complete.

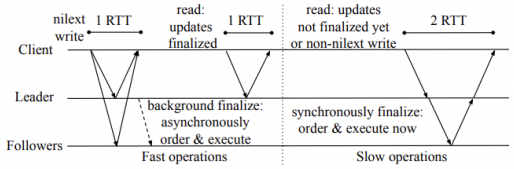

Figure 1: *The figure shows how a nilextaware replication protocol handles different operations.*

Skyros protocol is a leader-based protocol that differs in its operation depending on the type of request being made. The protocol suggests that durability, ordering, and execution need not always be entangled together. It proposes deferring ordering and execution for specific requests while ensuring safety. Figure 1 summarizes the protocol in both the common case and the uncommon case. In brief, for requests that don't require externalizing state to the client (nilext operations such as write), Skyros allows for faster processing times (1 RTT in most cases) by separating ordering from durability, which is accomplished by enabling an unordered durability log on all servers. Skyros requires a supermajority of replicas to respond to the client to achieve a 1 RTT protocol. Ordering and execution of these operations can either be done asynchronously in the background or when a system state needs to be externalized. However, while some operations (non-nilext operations such as read) do externalize the state, not every non-nilext operation on an object triggers synchronous ordering. In many workloads, updates to an object can be ordered and executed in the background before there arises a need for applications to externalise the state of the object. Therefore, the operations take a 1 RTT path in the fast path and a 2 RTT path in the worst case. Nilext operations, on the other hand, always take a 1 RTT path, thereby decreasing the latency for such operations and increasing performance.

3 Background and Motivation

In large storage systems, data is often replicated and partitioned across multiple data centers to ensure availability and fault tolerance. Nil-externality is a construct in storage interfaces that refers to operations whose effects are not visible to the client or the outside world until necessary. Skyros[2] is a replication protocol that exploits nil-externality and improves performance by deferring the execution of these operations and performs ordering and execution in the background. However, Skyros only allows reads from the leader, which can lead to performance bottlenecks. Paxos Quorum Reads [11] protocol addresses the issue of leader bottleneck for read operations. Paxos Quorum Reads enables reads from other nodes in the system and excludes reading from the leader. By integrating Paxos Quorum Reads with Skyros, we aim to improve read latency by providing for scalable reads in Skyros.

In many leader based-protocols, since the leader has the latest knowledge of the entire system, read operations made by the client are processed through the leader (to maintain linearizability) causing a single leader bottleneck as it has many messages to send and receive. If the read requests are scaled up by a magnitude, read requests directed to the leader would result in leader becoming the bottleneck for serving read requests, thereby underutilising the bandwidth on other replicas. This is true for all systems that use Paxos, Raft or similar variants for serving read requests. Read operations only require a value to be returned and in none of these protocols there exists a notion of differentiating 'read' operations from other operations. Therefore, in all scenarios read operations take a 2 RTT path.

Another drawback of protocols which heavily rely on the leader is that these protocols are not suited for a geo-replicated setting. For instance, if the clients are in one region and the leader is located in a geographically far away region, both throughput and latency for both reads and writes will suffer significantly even if the majority of replicas are in the same region as the client.

To address the leader bottleneck and enhance the performance of read operations, there are various potential solutions. For example, in systems like Zookeeper [10], reads can be carried out on any replica. However, this replica may not have yet received the latest update from the leader, resulting in the client receiving outdated information. Alternatively, reads can be performed on a quorum of replicas, but this does not resolve the problem of stale reads since the leader may be ahead of the replicas and not included in the quorum set. In systems that use quorum replica protocols such as Cassandra [5], they cannot guarantee strict linearizability and only offer eventual consistency. In Cassandra this is also accompanied by a read-repair phase that is performed in the background to fix stale reads by updating the log at all replicas.

Paxos Quorum Reads (PQR) [11] is a faster protocol that eliminates the leader bottleneck problem by completely avoiding reads at the leader. Figure 2 shows the common case operation of the PQR protocol. In the typical scenario, it operates in two phases - quorum read phase and rinse phase. Firstly, it carries out a quorum read phase by obtaining the highest "accepted" slot of the log from a quorum of replicas. The accepted value need not yet be executed on all the machines. Secondly, it performs a rinse phase where the client waits for a single replica node to execute the highest slot it received. Once it is executed, the corresponding slot value is returned to the client. Even though the leader bottleneck is eliminated, this is still a 2 RTT protocol. As an optimization to this protocol, if a Paxos-based system has made no progress, the last accepted slot is the same as the last executed slot. In these instances, the rinse phase can be entirely avoided, allowing the protocol to complete reads in 1RTT. The performance evaluation reveals that the PQR protocol is more effective for write-heavy workloads than for read-heavy ones. This is because the system can now push the overall throughput write throughput through the leader (using original Paxos) as the reads do not go through the leader. This solution inherently also solves the problem of a geo-replicated setting since leader is no longer present in the critical path for reads.

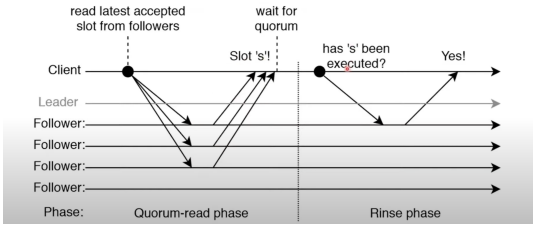

Figure 2: *The figure shows the two phases in the PQR protocol. In the first phase - the client sends a request to quorum of replicas and not just the leader. In the rinse phase, the client waits on a single replica and returns as soon as the slot is committed on the replica.*

The aim of this project is to investigate the advantages of a system that merges Skyros with Paxos Quorum Reads (PQR) to enable Skyros to handle scalable reads. Our objective is to build a system that utilizes Skyros, which is a 1RTT protocol for writes, and PQR, which is a 1RTT protocol for reads in the best-case scenario and a 2RTT in the worst-case. The resulting system is efficient for ready heavy and mixed workloads in a georeplicated setting and also eliminates the bottleneck caused by the leader in all types of workloads.

4 Design

This section provides a detailed explanation of the design of Syros, including its write and read path, as well as the importance of two additional variables, namely last_executed and last_accepted, for ensuring external consistency and linearizability. Inspired by Paxos Quorum Reads, Syros protocol

tracks these variables for every object at every replica that processes a corresponding request.

The last_executed variable keeps track of the highest sequence number of a request that has been executed by a replica for a given object, while the last_accepted variable records the highest sequence number of a request that has been accepted but not yet executed by a replica for that object. These variables are crucial for ensuring up-to-date reads and preventing stale reads for an object and ensuring that a request sees the effects of every successful preceding request.

4.1 Syros Write Protocol

As previously explained in the background, Skyros decouples durability from ordering and execution, which means that the write (or nilext operation) path is only concerned with durability, allowing write requests to be served in a single round-trip time (1RTT). Syros's write protocol preserves this approach and remains the same as Skyros's, maintaining its unique properties and serving writes in 1RTT.

1. The client sends a request to all replicas, which is uniquely identified by an ID.

2. Upon receiving a client request, a replica either appends it to the durability log (if not already appended) and sends an acknowledgement or simply sends an acknowledgement (if already appended). This avoids processing of duplicate requests.

3. Client waits to receive supermajority of acknowledgments from replicas. The client always ensures that the leader is a part of the supermajority set, as this is essential for ordering. the waiting period has a timeout.

4. On failing to receive supermajority of acknowledgments within the timeout, client retries with the same ID.

4.2 Background Ordering

By ensuring that the leader is always included in the supermajority set for all requests guarantees that it has all updates in its durability log in real-time order. Thus, the leader initiates an ordering protocol (such as VR) either periodically or when a non-nilext operation is issued by a client. Syros's background replication largely remains unchanged from Skyros's. However, it introduces two new variables for each object: last_accepted and last_executed. Below are the steps that occur when a leader initiates an ordering protocol:

1. The leader sends a PREPARE message for the updates in its durability log to all the replicas, including itself.

2. The replicas on receiving the PREPARE message, add it to their respective consensus logs, increment the last_accepted variable (initially set to 0) corresponding the object by one, and send a PREPAREOK acknowledgment to the leader.

3. The leader on receiving majority of acknowledgments, applies/executes the update and sends a COMMIT message in the background asynchronously to all replicas. The leader also asynchronously removes the update from its durability log.

4. The replicas on receiving the COMMIT message for an update, apply/execute it and increment the last_executed variable (initially set to 0) corresponding to the object by one. The replicas further remove the update from their durability log.

Therefore, by following this approach, the last accepted and last executed variables for an object are updated only for nilext operations pertaining t an object.

4.3 Syros Read Protocol

The Syros's read protocol is influenced by Paxos Quorum Reads, as it executes read requests at the followers instead of the leader. The initial steps of the read protocol are as follows:

1. The client sends a read request to all replicas.

2. Upon receiving the read request, the leader node does not process the request.

3. The follower nodes, upon receiving the read request, respond by sending back the requested object's value, set a flag to true to indicate that the object is present in the durability log, is present as well as the last accepted and last executed values corresponding to the requested object.

4. The client waits until it receives acknowledgments from a majority of replicas.

Further steps taken by the client to complete the read protocol are elaborated in subsequent sections.

4.3.1 Syros Fast Read Path

Once the client receives acknowledgments from a quorum of replicas, it retrieves the highest last_accepted value from all the read replies. If the highest last_accepted value matches the last_executed value, and the object key is not present in any of the follower durability logs, it indicates that the returned value accurately reflects all previous successful updates. The client can then consider this value as the final value. This approach enables reads to be served in a single round trip time (1RTT).

4.3.2 Syros Slow Read Path

If the highest last_accepted value from the read operation does not match the last_executed value or the key is present in the durability log, the client initiates the rinse phase. During the rinse phase, the client resends a read request to the follower with the highest last_accepted value and checks if the last_executed value matches the last_accepted value. This process can take two round-trip times (2RTT) or more to complete.

However, there may be situations where a follower node participated in the PREPARE phase but did not receive a COMMIT message from the leader. This could be due to several reasons such as the leader not receiving a majority of PREPAREOK replies or leader failure. To handle such cases, the last_accepted value is reset to the last_executed value during a viewchange process

5 Correctness

The correctness of Syros's read protocol can be understood by examining the Skyros and PQR protocols, as Syros utilizes aspects of both in its implementation for independent operations. The only difference between Syros and PQR is that Syros enters a rinse phase if an update is present in the durability log. Any supermajority and any majority sets always intersect at least one node as the sum of supermajority and majority is always greater than the number of nodes. Therefore, if an update is stored in the durability log, it will always be reflected in at least one of the majority replies received by the client. As a result, the protocol remains correct as the client waits for the update in the durability log to be applied or executed through the rinse phase.

6 Implementation

We built Syros by implementing our modifications on top of the open source Skyros protocol. To assess the performance of Syros in comparison to Paxos, we adopted viewstamped replication (VR) as our benchmark. VR is designed to tolerate up to f failures in a distributed system with $2\mathbf{f} + 1$ replicas. It operates on a leader-based protocol and progresses through a sequence of views, with one leader elected for each view in a round-robin fashion. VR implementations ensure linearizability, meaning that operations are executed in the order they appear in real-time, with each operation reflecting the effects of the ones that completed before it. Both Syros and Skyros preserve these properties, namely, availability, leader-based operation, and linearizability. In order to enhance the throughput, Syros maintains the batching mechanism used in the Skyros protocol. Figure 3 and Figure 4 provide a concise overview of the additional functionality that was developed and incorporated into the Syros protocol.

Additional Upcalls into Storage System:

- `UpdateLastAcceptedMap`: Increment value for object key in the *`LastAcceptedMap`* upon receiving PREPARE message.
- `UpdateLastExecutedMap`: Increment value for object key in the *`LastAcceptedMap`* upon receiving COMMIT message.
- `SetIsDurable`: Set the flag to true if the object key is in durability log.

Figure 3: *This figure illustrates the additional upcalls that the replication layer makes compared to a basic Skyros protocol.*

Additional Function to Client Interface

- `InvokeRinsePhase`: sent to only one follower, when the last_executed does match the highest last_accepted.

Figure 4: *This figure illustrates the additional function added to the client interface compared to a basic Skyros protocol.*

7 Evaluation

To evaluate Syros, we ask the following questions.

1. How does Syros perform in throughput compared to Paxos and Skyros on YCSB workloads in a geo-replicated setting?

2. How does Syros perform in throughput compared to Paxos and Skyros on YCSB workloads in a single region setting?

3. How does Syros perform in throughput compared to Paxos and Skyros on Uniform and Zipfian workloads?

Setup: We run our experiments on 5 replicas. Each replica runs on c220g2 CloudLab [12] machines. Each machine has two Intel E5-2660 v3 10-core CPUs at 2.60 GHz, 160GB ECC Memory (10x 16 GB DDR4 2133 MHz dual rank RDIMMs) and a Dual-port Intel X520 10Gb NIC (PCIe v3.0, 8 lanes). The clients also run on a machine with the same configuration but is not co-located with any of the replicas. This is done intentionally to simulate a real world scenario where the clients may not be located in the same data center as the replicas. Clients are closed loop and we increase the clients in order to slowly saturate the servers to their limits. Also, we have used UDP as the underlying transmission protocol.

7.1 Performance in Geo-Replicated Setting

We measure the p99 and average latencies for YCSB A,B and C workloads in a geo-replicated setting. Here, the leader of the group is located in Paris and other replicas are located in

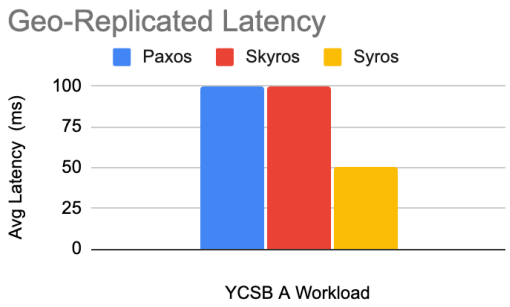

Figure 5: *Avg Latency in milliseconds for YCSB A workload in a geo replicated setting*

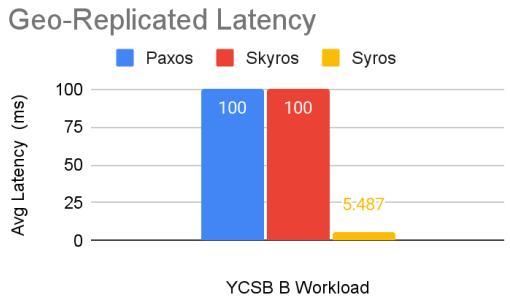

Figure 6: *Avg Latency in milliseconds for YCSB B workload in a geo replicated setting*

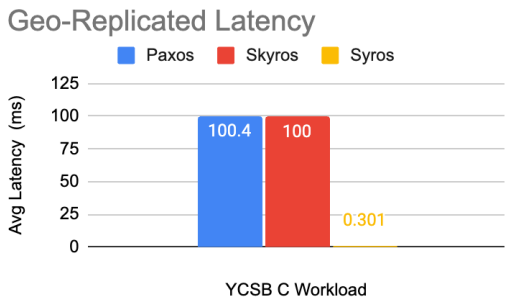

Figure 7: *Avg Latency in milliseconds for YCSB C workload in a geo replicated setting*

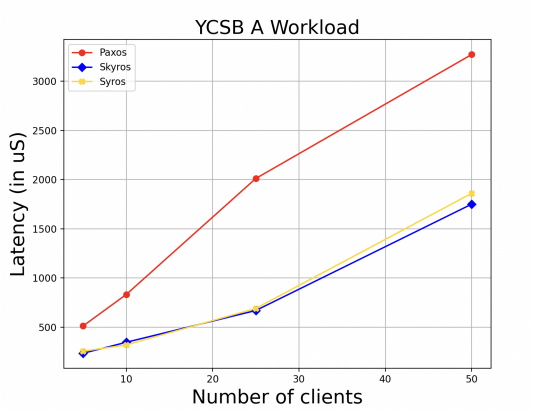

Figure 8: *Latency vs Number of Clients for YCSB A Workload*

Wisconsin. The ping latency between Wisconsin and Paris is approximately 96.87ms.

We observe from Figures 5 , 6, and 7 that Syros outperforms both Paxos and Skyros in the georeplicated settings for all the three workloads run - YCSB A, YCSB B and YCSB C. We measure average latency in milliseconds across the three protocols. Syros performs 2 times better, i.e half the average latency of both Skyros and Paxos. This is attributed to the fact that, in our geo replicated setup, the leader is not co-located with the other replicas and the clients. The leader is set up in Paris with a ping latency of approximately 96.87ms. In the read path, both Paxos and Skyros contact the leader, which causes their overall latency to be approximately equal to the ping latency between the client and the leader. This is due to the fact that the performance of the system is bottlenecked by the latency to reach the leader. However, Syros directly contacts the replicas instead of the leader.

We must observe however, that Skyros and Paxos perform similarly in a geo replicated setting. The advantages of background replication of Skyros are nullified as the latency to reach the leader is more than the background replication rate.

7.2 Performance in a Single-Region Setting

We present results on YCSB A workload, which represents a 50-50 read write ratio. We can see from Figures 8 and 10 Syros performs better than Paxos giving us lower latency and higher throughput with increasing number of clients. However, the performance of Syros is identical to that of Skyros in these cases. This is attributed to the fact that during the write path, Syros follows Skyros and write performance is identical. The YCSB A workload is a uniform workload, where every key is selected uniformly at random. As every key is equally likely to get selected, in the read phase, Skyros sees mostly 1RTT reads from the leader due to fast background replication. Most of the read request keys, fortunately need not be ordered. Similarly, Syros also sees 1RTT reads from the followers due to fast background replication. This makes their p99 latencies comparable. However, we see a slight drop in throughput of 15% from Skyros. This can be explained by the read protocol of Syros. In the read protocol, Syros clients wait for a majority of the replicas to respond back, before serving the read response to the client. This is not so, in the case of Skyros which is performing a fast 1RTT directly from the leader alone. This causes a slight drop in throughput when compared to Skyros. We can however verify

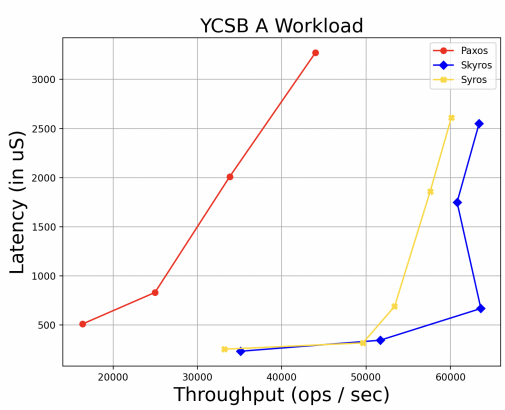

Figure 9: *Latency vs Throughput for YCSB A Workload*

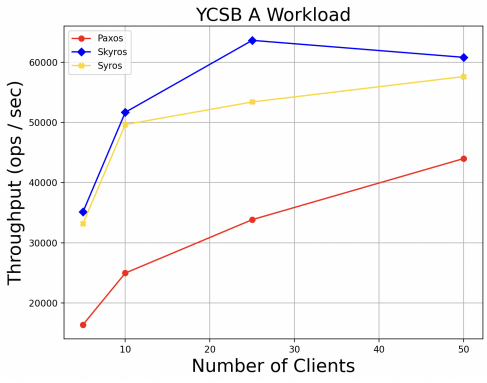

Figure 10: *Throughput vs Number of Clients for YCSB A Workload*

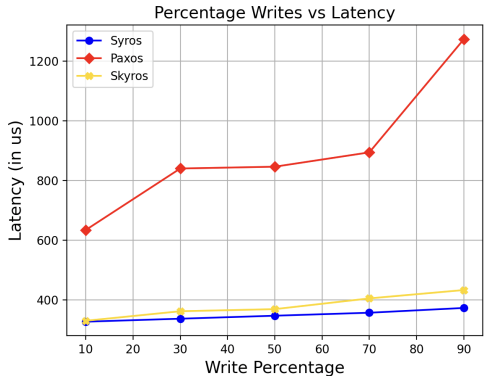

Figure 11: *Latency with increasing write percentage for uniform workloads*

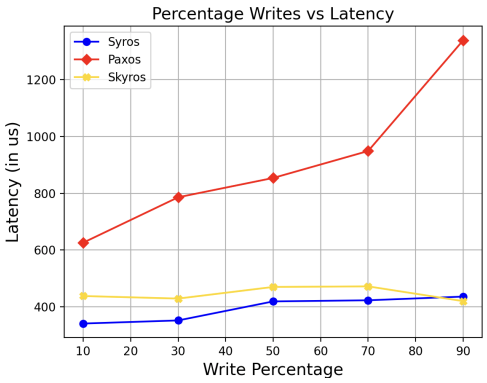

Figure 12: *Latency with increasing write percentage for zipfian workloads*

that Syros outperforms Paxos by giving 50% drop in latency and 58% increase in throughput. Figure 9 shows the latency vs throughput characterestics curve for Syros.

7.3 Performance in Uniform and Zipfian Workloads

We measure the p99 and average latencies of the requests in a single region setting but with increasing write percentages. We perform the same experiment on a uniform distribution of keys in Figure 11 followed by a zipfian distribution of keys in Figure 12 We see similar trends of latency vs write percentages for both distributions. Syros latency is slightly worse than Skyros in both uniform and zipfian distributions.

8 Future Work and Limitations

The main drawback in the Syros protocol as observed in the results is the slight performance dip compared to Skyros in uniform workloads. In continuation, we would like to explore other optimizations to reduce this latency. We can also benchmark other scenarios of geo replicated settings where a majority of clients are places in a different region than the clients and observe performance.

9 Conclusion

By avoiding heavy computation overhead, Syros can efficiently process reads from followers while maintaining consistency and linearizability. The last_executed and last_accepted variables track the sequence numbers of executed and accepted requests for each object at every replica, preventing stale reads and ensuring that all replicas process requests in the same order.

Syros's ability to handle reads from followers without sacrificing consistency or linearizability makes it an efficient and scalable solution. This feature is particularly useful in geo-replicated settings where the leader is farther away from the client than the supermajority of followers. As a result, Syros is an effective and scalable solution that can be applied to a variety of use cases and applications.

References

[1] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Strong and efficient consistency with consistency-aware durability. https://www.usenix.org/system/files/fast20-ganesan.pdf.

[2] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Exploiting nil-externality for fast replicated storage. https://ramalagappan.github.io/pdfs/papers/nilext.pdf.

[3] Iulian Moraru, David G. Andersen, and Michael Kaminsky. Paxos quorum leases: Fast reads without sacrificing writes. http://www.cs.cmu.edu/~imoraru/papers/qrl.pdf.

[4] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. https://homes.cs.washington.edu/~arvind/papers/specpaxos.pdf.

[5] Avinash Lakhsman and Prashant Malik. Cassandra - a decentralized structured storage system. https://www.cs.cornell.edu/projects/ladis2009/papers/lakshman-ladis2009.pdf.

[6] Mostafa Elhemali, Niall Gallagher, Nicholas Gordon, Joseph Idziorek, Richard Krog, Erben Mo Colin Lazier, Akhilesh Mritunjai, Tim Rath, Doug Terry, and Akshat Vig. Amazon dynamodb. https://www.usenix.org/system/files/atc22-elhemali.pdf.

[7] Leslie Lamport. Paxos made simple. https://lamport.azurewebsites.net/pubs/paxos-simple.pdf.

[8] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. https://web.stanford.edu/~ouster/cgi-bin/papers/raft-atc14.

[9] Barbara Liskov and James Cowling. Viewstamped replication revisited. https://pmg.csail.mit.edu/papers/vr-revisited.pdf.

[10] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. https://www.usenix.org/legacy/event/atc10/tech/full_papers/Hunt.pdf.

[11] Aleksey Charapko Ailidani Ailijiang and Murat Demirbas. Linearizable quorum reads in paxos. https://www.usenix.org/system/files/hotstorage19-paper-charapko.pdf.

[12] CloudLab. Cloudlab. https://docs.cloudlab.us/hardware.html.