INTERNATIONAL SCIENTIFIC JOURNAL OF ENGINEERING AND MANAGEMENT (ISJEM) ISSN: 2583-6129
VOLUME: 04 ISSUE: 09 | SEPT – 2025 DOI: 10.55041/ISJEM05020
AN INTERNATIONAL SCHOLARLY || MULTIDISCIPLINARY || OPEN ACCESS || INDEXING IN ALL MAJOR DATABASE & METADATA

Using ZooKeeper for ACL Propagation in Near Real Time

Nikhita Kataria
nikhitakataria@gmail.com

Abstract—Access Control Lists (ACLs) are a critical component for enforcing security as well as governance of data in a cloud environment. This necessitates the need for ensuring ACL changes can propagate quickly and reliably across infrastructure stack so that services can quickly act on any ACL change. In this paper we explore how ZooKeeper; a distributed coordination service can be leveraged to enable near real-time propagation of ACLs. We will present a high-level architecture, benchmarks with experimental setup and failure modes for ACL distribution using ZooKeeper. This paper will also discuss scalability, consistency and security models required for handling ACL changes in a robust fashion.

Keywords— ***ACLs, ZooKeeper, distributed systems, real-time security, infrastructure, configuration propagation, consistency, coordination services, RBAC, security auditing***

I. INTRODUCTION

In modern infrastructure platforms which are designed to automatically converge based on certain user patters, operate across thousands of geographically distributed machines and microservices rely on ACLs to enforce principle of least privilege across diverse set of resources including compute, storage and network. However, ACL propagation can be a challenging problem to solve if it's expected to be fast, reliable and if access needs to be updated across a distributed set of services. Traditional mechanisms such as proactive distribution of static rules or polling based mechanisms are often slow or unreliable under certain network factors that may cause system failures. Apache ZooKeeper offers an alternative with its low-latency coordination functionality and at the same time offering strong consistency guarantees. In this paper we explore the design and implementation for a ZooKeeper -based ACL propagation system that aims to provide real-time updates with low latency, strong consistency, fault-tolerance, secure handling and auditing of ACL changes. We will explore how ZooKeeper's atomic broadcast and watcher-based distribution proves to be instrumental in achieving aforementioned guarantees.

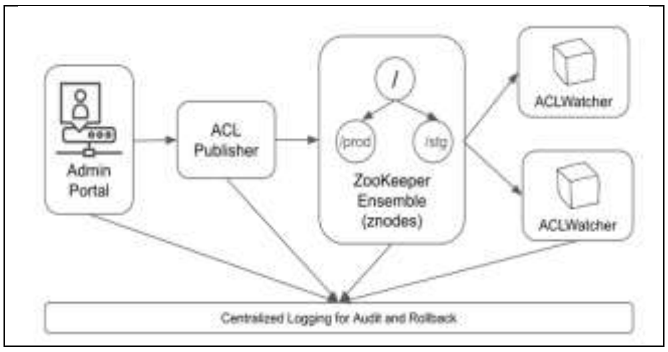
In today's world of growing automated services, ACL propagation at a scalable level also needs to comply with that organizational constraints such as struct compliance requirements by a few systems having a high security profile or services hosting multiple tenants that should not have access to each-others data. In this paper we will explore how ZooKeeper's propagation model creates a reliable, auditable, and extensible process for sensitive security configuration in cloud-native environments.

II. BACKGROUND AND MOTIVATION

Motivation to use ZooKeeper for ACL distribution stems from the fact that even a perfectly crafted ACL is not effective if it is not distributed in real-time across all the necessary systems. The boundary of security is only as strong as the weakest link in the system. If ACL updates lag or fail, this may cause services or users to have unauthorized or extended access to resources or deny legitimate access affecting security and availability. Some examples of real-world scenarios that may get impacted by ACL lag could be revoking access to a terminated employee, updating interservice authentication policies, emergency lockout of a compromised resource or granular permission changes due to security audits. ZooKeeper is a centralized service capable of maintaining configuration information while providing distributed synchronization. It provides a hierarchical namespace, strong consistency via an atomic broadcasting protocol (ZAB), watches for event notifications when `znodes` change and ephemeral nodes for session-based states. In current industry multiple multi-national companies already rely on ZooKeeper for metadata distribution, service discovery and very commonly for leadership election.

III. HIGH LEVEL ARCHITECTURE

This section presents high level architecture and involves components with a set role across ACL authorship, propagation, enforcement and observability around ACL distribution. First, an admin portal to serve as an interface for security administrators to manage CRUD operations for ACLs. Second, a publishing engine that validates and publishes ACLs to ZooKeeper. Third, ZooKeeper ecosystem with coordination and consistency engines. Fourth, an ACLWatcher in the form of an agent deployed on the same nodes where ACLs need to be applied which will be acting as a client to receive notifications about changes to the ZooKeeper state. This modular architecture is the key to ensuring scalability, low-latency, clear separation of concerns and low mean time to detect issues. Each component can evolve independently of each other; Admin portal for UX improvements, publishing engine for validations, ZooKeeper for distribution or schemas and ACLWatcher for execution and enforcement. This design would also allow schema changes that may need to happen in the future and allow creation of context-aware ACLs and even machine learning–enabled anomaly detection. This architecture separates concerns across ACL authorship, propagation, enforcement, and observability.


Admin Portal
ACL Publisher
/
/prod
/stg
ZooKeeper Ensemble (znodes)
ACLWatcher
ACLWatcher
Centralized Logging for Audit and Rollback
Figure 1. High level architecture for ACL distribution

A. *Admin Portal*

Admin portal will be a user-friendly web interface for security administrators or infrastructure operations with administrative access to manage the lifecycle of ACLs. It would support capability to Create, Read, Update and Delete ACLs with role-based access control to ensure that even an admin operates only on allowed scope. Audit logs and version tracking on the UI should be embedded to provide accountability, traceability and complication information in accordance with the organizational security policies. Audit logs, front-end validation and version tracking will be embedded in the portal to provide traceability.

B. *Publishing Engine*

Once ACL changes are validated and submitted through admin portal, publishing engine's role will be to translate these definitions into a structured and versioned format. As an example, if source and destination in an ACL rule have IP addresses with subnet masks, this engine would validate the entries for syntactic as well as semantic correctness such as invalid CIDRs, overlapping rules or any missing metadata. It will then create individual rules and publish this data to a set of designated ZooKeeper `znodes`. This engine will surface a schematized log and version store ensuring that during break glass incidents changes can be audited and rolled back.

C. *ZooKeeper Ecosystem*

Apache ZooKeeper will act as the central coordination service in this architecture providing consistent, fault-tolerant and replicated storage for ACL states across the entire stack. The hierarchical namespace (`znodes`) will be used to organize ACLs by namespaces, services and host types. In this paper, we organize ACL entries in ZooKeeper `znodes` via meaningful keys such as ACL ID, tenant name or environment. As an example, we create following policies: `/acl-policies/web-prod,/acl-policies/web-stage, /acl-policies/db-prod and /acl-policies/db-staging` where each `znode` contains ACLs relevant to its context. Data in each `znode` will be structured using a JSON or YAML format to ensure ease of parsing and in-place updates. These entries capture Rule ID, source and destination IP with subnet mask, transport protocols, source and destination port ranges and intended action such as allow or deny. Advanced models may also encode metadata such as TTL, audit source, or workflow ID for traceability. Table 1. presents example entries for znode `/acl-policies/web-prod` version 1. This structured representation allows for easy readability and automated enforcement by downstream agents such as ACLWatcher.

```
{
  "version": "1.0",
  "rules": [
    {
      "id": "rule-1",
      "src_ip": "10.1.0.0/16",
      "dst_ip": "192.168.100.0/24",
      "protocol": "tcp",
      "port": "443",
      "action": "allow",
      "comment": "Allow HTTPS from internal net
to web frontend"
    },
    {
      "id": "rule-2",
      "src_ip": "0.0.0.0/0",
      "dst_ip": "192.168.100.50/32",
      "protocol": "tcp",
      "port": "22",
      "action": "deny",
      "comment": "Block SSH access from all to
jump box"
    }
  ]
}
```

Table 1. Example entries in Znode /acl-policies/web-prod

D. *ACL Watcher*

In any distributed system the actual consumer of ACL is the target nodes where services are deployed as they rely on updated ACLs to manage inter or intra service access. The ACL rules govern entries in local firewall rules or IP tables. ACLWatcher deployed on every host in the datacenter would monitor ZooKeeper znodes for changes and apply the respective ACLs locally by updating iptables, nftables, application configurations or system permissions. ACLWatcher would be classified as a security application i.e. require super user permissions to be able to update the afore-mentioned configurations and can be deployed as a systemd service, Docker sidecar, or Kubernetes DaemonSet depending on the environment.

IV. RELIABLE EVENT HANDLING

Access control lists tend to change often in response to operational, service and security changes. Since these changes happen in a distributed manner it is essential to record a timeline such that debugging and rollback decisions can be taken with confidence. In this architecture we introduced ACLWatcher as an agent running on all target nodes which observe changes however designing a reliable event-handling system around ZooKeeper requires addressing multiple limitations and caveats. First, Watchers are one-short notifications i.e. once a watcher callback is executed it is automatically removed and must be explicitly re-registered. This necessitates a need to ensure ACLWatcher immediately processes the notification and re-registers itself on the `znode` to ensure future events are not missed. Second, transient network, service or node failures can lead to temporary inability to watch or read from a `znode`. This is be tackled by implementing retry loops for watch registrations with exponential backoff to avoid overwhelming ZooKeeper while recovering for mass outages and ensure ACLWatchers are self-healing. Third, ACLWatchers might have missed events in case of client restarts and ensuring eventual consistency in convergent distributed systems is necessary. To solve this, we use periodic full reconciliation with ZooKeeper to validate local ACL state against the desired source of truth to ensure state drifts due to client-side bugs are eventually resolved. Since ACL changes are critical it is essential to have a service level objective (SLO) around it and in this paper, we propose a full sync every 5 minutes.

V. EXPERIMENTAL EVALUATION

Table 2. presents results from an experiment to determine how size of ACL entries would affect full sync time from publishing to ZooKeeper to ACLWatcher detection with 3-node ensemble running ZooKeeper 3.8.x on a virtual machine with 4 cores and 8GB Kubernetes node. Without any ZooKeeper failure or network delays, we observe that notification latency for ZooKeeper is largely independent of the ACL size and parsing time scales linearly with size of ACLs. Total sync time under 5 seconds for 10,000 rules suggests that near real-time propagation of ACLs via ZooKeeper is feasible if the ACLWatcher is optimized for performance as beyond 1000 rules we observe CPU usage for ACLWatcher increases exponentially. CPU usage increases linearly due to parsing of ACL rules and in this paper, we introduce versioning of ACL rules which optimizes

ACLWatcher by only parsing the rules that have changed. With versioned updates, ACLWatcher efficiently skips redundant updates, conserving CPU. One observation is that at 10,000 rules CPU usage may content with co-located workloads in a containerized environment justifying delta propagation.

ACL Size	Notification Latency (ms)	Cpu Usage (%)	Parse Time (ms)	Enforcement Time (ms)	Total Sync Duration (ms)
10	15	1.2	5	20	15
100	17	3.5	10	50	45
1,000	19	17.5	50	180	380
10,000	25	61.4	300	1600	3400

Table 2. Comparison of enforcement time with varying ACL sizes.

A. Experimental evalution with optimizations

Even with delta propagation large ACL sets might incur significant CPU overhead in environments where source and destination IPs are impacted by nodes where services run. Considering a Kubernetes based deployment, pods can move around nodes causing rule changes often. Even one pod movement would result in delta for ACL to change. To optimize further we apply ACL rule compaction by merging overlapping and redundant ACL entries in ACLPublisher before publishing to ZooKeeper and enable parallel rule application in ACLWatcher agents using thread pools. Table 3. Covers results with these variants and their impact on propagation for various ACL sizes.

ACL Size (Rules)	Baseline Avg CPU (%)	+ Delta Propagation Avg CPU (%)	+ Rule Compaction Avg CPU (%)	+ Parallel Rule Application Avg CPU (%)
10	1.2	0.8 (-33%)	0.7 (-12.5%)	0.6 (-14.3%)
100	3.5	2.3 (-34%)	2.0 (-13%)	1.7 (-15%)
1,000	17.5	11.6 (-34%)	10.0 (-14%)	8.5 (-15%)
10,000	61.4	42.0 (-32%)	36.0 (-14%)	31.5 (-12.5%)

Table 3. CPU Usage based on optimization with delta propagation, rule compaction and parallel rule execution.

B. Experiment evaluation with failures

Failures could comprise of multiple scenarios such as network issues, session losses, node restarts, ACLWatcher crashes or transient read or write failures triggering retries. Retries often will have an impact on CPU usage causing it to increase due to extra work on retry attempts specially if exponential backoffs are used due to additional timer and recovery logic. Table 4. and Table 5. present the impact of retries on the same setup post multiple optimizations where every optimization might also turn into a CPU overhead.

ACL Size (Rules)	Baseline Avg CPU (%)	Baseline + Failures (%)	Delta Propagation Avg CPU (%)	Delta Propagation + Failures (%)
10	1.2	1.8 (+50%)	0.8	1.1 (+38%)
100	3.5	5.0 (+43%)	2.3	3.3 (+43%)
1,000	17.5	27.0 (+54%)	11.6	18.5 (+59%)
10,000	61.4	93.0 (+51%)	42.0	63.0 (+50%)

Table 4. Baseline and Delta Propagation with Failures

ACL Size (Rules)	Rule Compaction Avg CPU (%)	Rule Compaction + Failures (%)	Parallel Rule Application Avg CPU (%)	Parallel Rule Application + Failures (%)
10	0.7	1.0 (+43%)	0.6	1.2 (+100%)
100	2.0	3.0 (+50%)	1.7	2.7 (+59%)
1,000	10.0	16.5 (+65%)	8.5	13.5 (+59%)
10,000	36.0	55.0 (+53%)	31.5	48.0 (+52%)

Table 5: Rule Compaction and Parallel Rule Application with Failures

C. Conclusion

In this study, we explored how optimization of ACL updates with delta propagation, rule compaction and parallel propagation make a big difference in resource usage which also impacts the propagation latency as a system with high CPU usage would cause resource contention impacting overall performance. We also observe that while failures occur, CPU use spiked due to retries demonstrating the tradeoff between performance and resilience. Overall, with optimizations we could with confidence conclude that ACL propagation can be achieved in near real-time via ZooKeeper.

VI. SECURITY MODEL

ZooKeeper out of the box supports authentication mechanisms using SASL such as Kerberos or plain digest-based authentication. Znode ACL's define permissions on who can do what on a specific `znode` and in the architecture described in this paper only ACL publisher engine would be allowed to have write access to ZooKeeper ensemble. ACLWatchers would be only allowed read access on the port that would be listening on. Also, the ZooKeeper ensemble itself is secured using TLS with host and network level access embedded. Optionally `znode` payloads can be encrypted using per-tenant keys obtained from a centralized Key Management Service (KMS). Experiments in this paper do not encrypt the payloads and most certainly having this capability would result in increased latency due to encryption and decryption time. In terms of auditing, every change to a `znode` is logged with timestamp, user or service initiating the change and delta different from the previous value. These logs can be maintained in a distributed logging service as per the compliance requirements of the environment. Each ACL entry in the ensemble may include a TTL field to automatically expire permissions after a set period-of-time. ZooKeeper offers a versioning model that supports point in time rollback using historical values that can be used in case of outages. These controls overall help meet the compliance and security requirements for propagating ACLs.

VII. CHALLENGES AND MITIGATIONS

ZooKeeper has been adopted for multiple use cases across industry for coordination and configuration management however it comes with its own limitations. While designing ACL storage and propagation mechanics scale is of utmost importance and one limitation is number of watches that can set per `znode`. First, since each watch consumes CPU and Memory resources relying heavily on `znodes` can quickly hit scalability issues. Recommendation to solve afore-mentioned scenario is to store granular paths. For example, increase of placing all ACLs under /acl-policies/, split the path into multiple buckets based on tenant thus distributing the watches and reducing load on a single znode. Second, frequency of writes can overwhelm ZooKeeper ensemble, leading to degraded performance or consistency issues. A practical approach is to implement a batching logic which prevents rapid successive writes and batches multiple writes into a single operation with a minimum period as it is important to ensure that propagation stays near real-time. Third, over a period there is a risk of having a bloated JSON payload stored in a Znode leading to debuggability issues, significant memory consumption that further causes slowness in read/write operations. A layman solution to circumvent this is to have predefined ACL templates to avoid duplication and strike a balance between having too granular and too broad ACLs. An advanced approach is to use compression of payload before storing to keep the `znode` size manageable. However, any compression may result into latency increases and resource overhead and hence the algorithm used must be simple enough to minimize impact and advanced enough to provide meaningful efficiency gains. Finally, failures and partial updates need to be accounted for. Consumers may miss critical updates due to transient network issues, session expirations, or simple service restarts. Such inconsistencies require a period full sync possible using sequence numbers to detect missed updates. These strategies combined, help maintain ZooKeeper's stability and reliability while enabling it to support complex access control use cases at scale.

VIII. LEARNINGS

In the previous sections, we presented various experiments that outlined ripple effect ACL failures can have on other layers of infrastructures which yielded several learnings. First, ACLs were responsible for more than 30% of the deployment issues where either deployment infrastructure had issues or applications had runtime issues highlighting how ACLs can influence stability of the infrastructure pipeline. Second, troubleshooting took 4X longer as ACL related issues were far more time consuming to identify and resolve than code or configuration issues due to their indirect nature of impact. Thirst, security was often seen being sacrificed for speed where 30% of the issues are resolved by broadening the permissions as a quick fix; a trade-off that highlights the operational pressure of making applications just work. These findings validate that ACLs if poorly managed do not just create isolated failures but have cascading impact on reliability and at times promote security risks. They also highlight how important it is to invest in observability and tooling to operationalize ACLs.

IX. CONCLUSION

ZooKeeper provides a strong foundation for near real-time ACL propagation. With a streamlined architecture publishing and watching changes, versioned updates, and a strong security model, systems can efficiently deploy ACL changes across fleets in near real time with very low latency. ZooKeeper's guarantees of consistency and ordering makes it very suitable for security-critical use cases as well. In this paper, we demonstrate through a distributed design and benchmarks that this approach can meet the operational demands of a modern distributed system. Organizations can use this pattern to improve response time for access changes, reduce human error in enforcement, and maintain auditable access control at scale.

X. REFERENCES

[1] Apache ZooKeeper Documentation, [Online]. Available: https://zookeeper.apache.org. [Accessed: Aug. 16, 2025].

[2] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for Internet-scale systems," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, Boston, MA, USA, Jun. 2010.

[3] LinkedIn Engineering, "How LinkedIn uses ZooKeeper for configuration management at scale," Nov. 2019. [Online]. Available: https://engineering.linkedin.com. [Accessed: Aug. 16, 2025].

[4] A. Chanda *et al.*, "Security propagation in distributed systems," in *Proc. ACM Conf. Computer and Communications Security (CCS)*, Seoul, South Korea, Nov. 2021.

[5] Netflix Tech Blog, "Managing service auth at scale," Jul. 2020. [Online]. Available: https://netflixtechblog.com. [Accessed: Aug. 16, 2025].

[6] AWS Security Blog, "Best practices for implementing ACLs and RBAC in cloud-native applications," Mar. 2021. [Online]. Available: https://aws.amazon.com/blogs/security. [Accessed: Aug. 16, 2025].

[7] Microsoft Azure Architecture Center, "Designing secure ACL and RBAC propagation models," 2020. [Online]. Available: https://learn.microsoft.com/azure/architecture. [Accessed: Aug. 16, 2025].

[8] Kubernetes Documentation, "Using ConfigMaps and Secrets for distributed configuration," 2023. [Online]. Available: https://kubernetes.io/docs. [Accessed: Aug. 16, 2025].

[9] A. Shamir, "How to share a secret," *Commun. ACM*, vol. 22, no. 11, pp. 612–613, Nov. 1979.

[10] Google Cloud Architecture Center, "Securing workload identity and access control in distributed environments," 2022. [Online]. Available: https://cloud.google.com/architecture. [Accessed: Aug. 16, 2025].

[11] F. P. Junqueira, B. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *Proc. IEEE/IFIP Dependable Systems and Networks (DSN)*, Hong Kong, China, Jun. 2011, pp. 245–256.

[12] Red Hat, "Securing applications with role-based access control (RBAC)," 2021. [Online]. Available: https://www.redhat.com. [Accessed: Aug. 16, 2025].