Enhancing Scalability and Performance in Distributed Systems: Analyzing Locking Mechanisms in HA Key-Value Stores

Nivardo A. L. Castro[1], Cidcley T. Souza[1], A. Wendell O. Rodrigues[1]

[1]Federal Institute of Education, Science and Technology of Ceará (IFCE)
Fortaleza – CE – Brazil

{nivardo.albuquerque06}@aluno.ifce.edu.br

{cidcley,wendell}@ifce.edu.br

Abstract. *Distributed systems are fundamental to the landscape of modern computing, serving as foundation of applications from large-scale cloud infrastructures to distributed databases. These systems face intricate challenges in maintaining data integrity and managing concurrent processes across decentralized environments. Among the mechanisms devised to navigate these complexities, distributed locking stands out as a pivotal strategy for orchestrating resource access among multiple nodes, ensuring operational coherence and data consistency. This paper addresses the mechanisms of distributed locking within the context of key-value storage systems, which are celebrated for their straightforwardness and high scalability. Our investigation encompasses an analysis of both blocking and non-blocking strategies for resource acquisition, enlightening the balance between securing exclusive access to resources and minimizing latency to enhance user experience and system efficiency. Additionally, we survey the scalability challenges that emerge as the system expands, evaluating how these mechanisms scale across an increasing number of nodes and operations. The study probes into performance bottlenecks that often manifest in distributed environments, identifying strategies to mitigate these constraints while maintaining high throughput and responsive systems. Moreover, we focus on the critical aspects of consistency and latency, exploring architectural and algorithmic solutions designed to harmonize the two, thereby facilitating a seamless and efficient distributed operation. Benchmarking evaluations are presented, incorporating metrics such as throughput, latency, and scalability, providing insightful findings that contribute to the broader understanding of distributed systems coordination, offering valuable guidance for system designers and developers.*

1. Introduction

Distributed systems are common in contemporary computing setups, spanning from large-scale cloud infrastructures to distributed database architectures. Ensuring data integrity and mitigating race conditions within concurrent systems are crucial in such environments [van Steen and Tanenbaum 2016]. Distributed locking mechanisms serve as fundamental tools to address these challenges by orchestrating resource access across multiple nodes.

Moreover, in a society that is progressively interconnected, where software applications range from vehicular monitoring [Echavarría et al. 2020] to large-scale video analysis [D'amato et al. 2021], the economic impact of system downtime is becoming more pronounced.e, with even brief periods of disconnection resulting in substantial financial losses for companies. To preempt and alleviate such repercussions, high availability configurations are implemented to enhance the fault tolerance of systems[Yeo et al. 2006].

This paper undertakes an investigation into the realm of distributed locking within the context of high available key-value storage systems. Key-value storage solutions represent foundational components within distributed applications due to their simplicity and scalability[Xu 2018]. The strategies here analyzed, allow developers to seamlessly integrate locking mechanisms into their current codebases, capitalizing on pre-existing components for enhanced efficiency. However, coordinating access to these stores within distributed settings introduces complexities associated with concurrency management and synchronization, specially when high availability is required.

The investigation navigates through the intricacies of distributed locking mechanisms, exploring both blocking and non-blocking strategies for lock acquisition. On the paper by Piotr Grzesik [Grzesik and Mrozek 2019], wide-spread key-value stores were evaluated, comparing performance based on time taken to acquire locks, each store with its own locking algorithm. This paper aims to extend Grzesik findings with new metrics and algoriths while shedding some light on correctness and violations.

Redis, etcd and ZooKeeper were store chosen to be analyzed based their usage on the industry and their underlying replication strategies and consensus algorithms, respectively, Sentinel [Redis 2024], Raft [Ongaro and Ousterhout 2014] and ZAB [Kamil et al. 2021].

2. Related Work

As aforementioned, Piotr Grzesik [Grzesik and Mrozek 2019] evaluated performance for multiple high available key-value stores, concluding that while Redis is fast on non-competing environments, its performances degrades on competing environments with high concurrency. It's also noted that etcd, ZooKeeper and Consul shared similar performance. Grzesik didn't explore violations and detection times.

The Jesper Team has also conducted several analysis on both performance and correctness on multiple key-value stores, finding inconsistencies in most of them. For etcd[Kingsbury 2020a], it was found major flaws on locking mechanisms and documentation, even after fixes it's not recommended to use etcd locks without fencing tokens. ZookKeeper was found to be more reliable, albeit some false negatives were found during partitions, where one write was not acknowledged but it was indeed committed[Kingsbury 2013b]. Redis was found to provide guarantees up to causal consistency, making it unsafe for locking purposes[Kingsbury 2013a].

While some research studies have delved into aspects such as the performance and correctness of individual non-blocking solutions, there is a gap in the literature regarding a comprehensive comparison of these solutions within the broader context of their correctness.

3. Locks and Leases

The foundational concept behind the lease mechanism involves a server issuing a token, referred to as a lease, to a requesting client. Upon granting a lease, both the client and server use their wall-clock time to determine the lease's expiration, based upon a predefined time-to-live (TTL). The introduction of expiration is crucial in preventing deadlocks in the event of client crashes while holding a lease. Despite its efficacy in addressing the issue of mutual resource access by restricting access to clients lacking an active lease, the reliance on well-behaved clocks[Gray and Cheriton 1989] make leases somewhat insecure and vulnerable to exploitation.

In the context of local locks, POSIX [Atlidakis et al. 2016] faced an analogous challenge with shared process locks. Should a process crash while holding the lock, it fails to release it, consequently culminating in deadlocks. This dilemma is addressed by robust locks, which ensure the release of locks upon the crash of the owning process. However, the usage of this solution is not possible in the realm of distributed systems; in this context, the assumption that a node has been disconnected may not hold true, presenting a characteristic known as Byzantine Fault[Driscoll et al. 2003], reinforcing the need for an lease timeout.

With this said, is safe to infer that the concept of distributed locking does not hold the same guarantees as local locks. From now on, the term lock is used in the paper interchangeably with lease, not implying same guarantees as operational system locks.

3.1. Fencing tokens

Martin Kleppmann introduced the concept of fencing tokens as a method to mitigate potential lease violations [Kleppmann 2017]. This approach is illustrated through a theoretical scenario in which two clients endeavor to obtain a lease and commit a state to a storage server, ensuring mutual exclusion. If a client that holds the lease experiences a "stop the world"interruption, the lease might expire without the client's knowledge. This can lead to a flawed state where the resource mistakenly becomes available to another client, while the original leaseholder continues operations under the false assumption that it still maintains control over the lock.

Fencing tokens are unique increasing tokens that are shared with the client that acquired the lease and must be passed to the resource, that should now check if the provided token it not bigger than the last processed token, denying the operation if the constraint is broken.

Notice that the resource must now implement that check, introducing a new point of failure and possible races on the application side if the resource is not able to do the checking atomically or in a atomic transaction.

With the limitations presented on section 3.1, let's now understand the inner workings of the chosen stores and some possible implications on acquisition and performance.

4. Stores

4.1. Etcd

Etcd uses raft[Ongaro and Ousterhout 2014] as it's consensus algorithm and at the writing of this paper its official documentation claims strict serializability [etcd 2024], which indicates serializability and linearizability, imparting the perception that each operation executed by concurrent processes seemingly occurs instantaneously within the interval spanning from its initiation to its completion[Herlihy and Wing 1990]. Etcd does not ensure linearizability for watch operations.

4.2. Redis

At the time of writing of this paper when ran in a Sentinel [Redis 2024] setup, Redis uses asynchronous replication, this means that at any given time data changes may be be lost if a partition happens on a master node that has not yet synchronized with the other nodes[Kingsbury 2013a]. An algorithm called Redlock has been proposed to allow redis to have a highly available setup when working with multiple master nodes, note that this seutp is not comparable in this paper since a multiple master setup does not have complete replication on the protocol; Still, the RedLock algorithm does not include the generation of fencing tokens, not allowing the mitigation strategy to be employed, furthermore the safety and correctness of it has been challenged by Martin Kleppmann [Kleppmann 2017];

4.3. ZooKeeper

Zookeper uses an consensus protocol focused on ordering of messages while being fault-tolerant and highly-available[Kamil et al. 2021]. It is based on ZAB, ZooKeeper's atomic broadcast protocol. Those are key feature while working with distributed systems coordination, guaranteeing order avoids deadlocks and makes blocking locks easier to implement.

5. Strategies for Distributed Locking and Issues

Blocking strategies, characterized by a client's waiting for access to a particular resource until it becomes available, effectively increases the changes of acquiring leases. However, this approach can engender bottlenecks and scalability challenges due to the potential for multiple clients contending for the same resource simultaneously. In contrast, non-blocking strategies prioritize reducing latency by allowing clients to proceed with other tasks while awaiting resource availability. This optimization comes at the cost of stringent execution guarantees, necessitating modifications to the codebase to handle potential errors and implement retry mechanisms in the event of resource unavailability. Thus, the choice between blocking and non-blocking strategies involves a trade-off between resource exclusivity and system responsiveness, with implications for both performance and code complexity.

In the realm of concurrency coordination, etcd natively supports a non-blocking mutual execution framework based on leases[etcd 2024]. This mechanism operates through the atomic comparison of revisions – monotonically incrementing version numbers assigned to keys – swapping them in case of success. The revision number may be used as a fencing token. The official Go [Authors 2024] library, also provides a native blocking method that uses watches to check for deletes on lock owners.

Parallel to etcd, Redis, within its Sentinel configuration, advocates for the employment of a straightforward compare-and-swap (CAS) strategy accompanied by expiration parameters. This approach involves the generation of incremental fencing tokens within the CAS transaction to facilitate non-blocking lease procurement. It is imperative to acknowledge, however, that this methodology exhibits limitations in partitioned environments due to the replication characteristics of Redis Sentinel, where even the generation of fencing tokens may become obsolete.

Redis team has been working on a new replication algorithm based on raft, Redis-Raft, the a work-in-progress version of the implementation has been tested by Jepsen and found to be promising [Kingsbury 2020b]. Redis-raft, once published, could fix issues that lead to the creation of Redlock, and support highly-available leases.

ZooKeeper documentation suggests one recipe for locks using watches – asynchronous notifications of node modifications. This recipe entails an initial attempt to secure the lock, succeeded by the observation of the locked key and, potentially, the subsequent key in the queue . With each modification notification, an attempt to acquire the lock is reiterated. Note that ZooKeeper allows setting a watch on the same transaction as getting the node, avoiding race conditions. It also returns a ZXID that can be used as fencing token.

Watch triggers are asynchronous, with ZooKeeper ensuring the dispatch of notifications to clients via underlying TCP communication. Given the absence of a znode expiration concept, ZooKeeper uses ephemeral nodes, which are automatically eliminated upon the disconnection of the creating session, thereby compensating for the lack of expiration functionality and avoiding deadlock scenarios. Nevertheless, network partitions may occasion the premature release and acquisition of leases by competing entities, while the original owner remains under the impression of continued ownership dilemma,this a different instance of the same problem presented on section 3.1.

To facilitate a comparative analysis of blocking and non-blocking strategies, the latter will be adapted to a blocking framework utilizing a retry-and-timeout technique. This approach entails repeated attempts to acquire the lock until a pre-determined timeout threshold is reached. Should the lock remain unsecured beyond the timeout duration, further attempts are ceased, culminating in failure. While this method mitigates the passive waiting, that lead to race conditions, it increases resource consumption.

6. Benchmarking

Tabela 1. Hardware Specification

OS	Ubuntu 18.04
CPU	Intel I7 8665U
RAM	16GB 2400MHz
Storage	120GB SSD Sata

For the benchmarking, a straightforward HTTP API was implemented to simulate a real-world resource. This API incorporates fencing tokens, as detailed in Section 3.1, and offers two operations:

- **Get Value:** a GET endpoint that retrieves the shared value, returning a `200 OK` response along with the current counter. This endpoint also has a 5% chance to sleep for 1s on every call. This is made to emulate “stop the world” freezes.

- **Set Value:** a PUT endpoint for setting the shared value, returning a `201` status code upon successful execution.

Both operations accept a fencing token as input; if the token is deemed incorrect, a `409 Conflict` error is returned. Notably, the API was intentionally designed to support concurrent requests and induce race conditions on counter operations, with the assurance that the fencing token will keep it consistent.

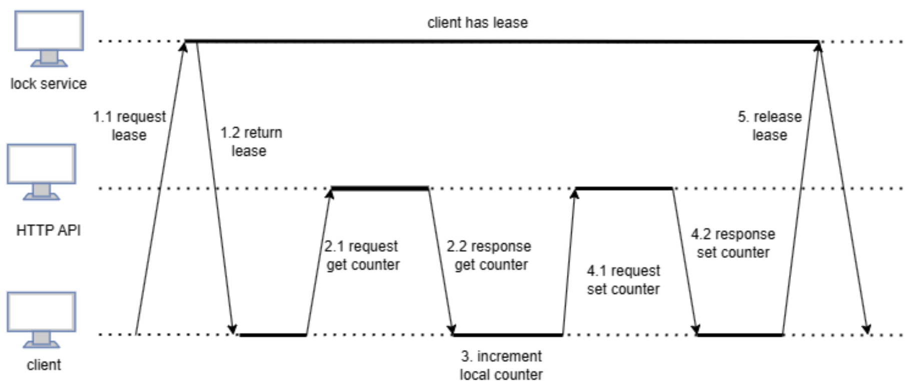

Figura 1. Steps for testing blocking strategies

The benchmark code was created following the steps illustrated in Figure 1:

1. Acquire a distributed lease with fencing token.
2. Issue a Get operation and save to the local counter.
3. Add 1 to the local counter.
4. Issue a Put operation passing the updated local counter.
5. Release the lease.

The rationale behind this decision stems from the creation of a racy counter. Should steps (2, 3, 4) be executed concurrently, there exists the risk of the counter losing increments. The usage of the distributed lease is intended to address this issue, also guarded by the fencing token.

For the test to account for high-availability, all the stores were run in a 3 node cluster setup. Partitions were forced every 10 seconds during the tests by force killing random nodes of the key-value stores, this should lead to new elections and probable issues with Redis.

The tests were run on a single machine with the hardware specifications outlined in Table 1. Multiple nodes were simulated using Docker. Each iteration of the test ran for 10 minutes.

A baseline benchmark was conducted without implementing any locking mechanisms to establish a reference point for assessing the API's performance under optimal conditions. This approach allows for a clear comparison between the unlocked state and various locking configurations. By excluding locks, the benchmark tests the system's maximum throughput and response time, providing a fundamental understanding of the API's capacity and efficiency. This baseline serves as a control in the experimental design, enabling subsequent analyses to quantify the impact of different locking strategies on the system's performance.

6.1. Performance Indicators

To comprehensively evaluate the performance of distributed locking mechanisms within the established testbed as detailed in section 6, two critical performance indicators were selected: Throughput and Conflicts. These metrics are essential for assessing the efficiency and robustness of distributed locks across a variety of operational contexts and stress scenarios.

Throughput High throughput indicates that the system can handle a significant number of operations simultaneously without substantial delays, reflecting efficient management of concurrent accesses. This metric is particularly important as it demonstrates the capability of the locking mechanisms to facilitate rapid processing while guaranteeing mutual exclusion.

Conflict Rate The conflict rate is a critical indicator of the reliability and correctness of the locking protocol. It measures the frequency of conflicts that arise when the locks fail to ensure mutual exclusion among concurrent operations. A high conflict rate is indicative of synchronization issues.

Tabela 2. Benchmark with 3 Clients

3 Clients				
Store	Lease TTL(s)	Total Ops	Op/s	Conflicts
Base Line				
N/A	N/A	37440	62.40	N/A
Redis	1	10389	17.31	761
ZooKeeper	N/A	10031	16.71	0
Etcd	1	5176	8.62	294

Tabela 3. Benchmark with 5 Clients

5 Clients				
Store	Lease TTL(s)	Total Ops	Op/s	Conflicts
Base Line				
N/A	N/A	57720	96.2	N/A
Redis	1	13894	23.15	801
ZooKeeper	N/A	10987	18.31	0
Etcd	1	7642	12.73	412

7. Result discussion

The comparative analysis revealed on Tables 2 and 3, show that distributed locking mechanisms across different HA key-value stores exhibit varied performance under high concurrency and partitioning scenarios. The benchmarks highlight the trade-offs between throughput, consistency, and conflict handling, essential for designing reliable distributed systems.

Redis, despite its popularity and ease of use, demonstrated limitations in handling partitions and maintaining data consistency under high load, as indicated by the number of conflicts observed. This aligns with previous studies questioning Redis's suitability for distributed locking without additional safeguards like fencing tokens.

Etcd, leveraging the Raft consensus algorithm, showed lower throughput but better consistency and fewer conflicts. Its design principles around strict serializability and linearizability suggest a stronger guarantee for distributed locking, albeit at the cost of performance, making it a potentially more reliable choice for scenarios where data integrity is paramount. Is also noted that Etcd, despite the lower throughtput, scaled well with the number of clients, achieving a 47% increase on op/s with 5 clients when comparing to 3 clients.

ZooKeeper's performance and conflict handling suggest its effectiveness in distributed locking, attributable to its use of ZAB for consensus and its recipe for locks using watches. Despite the weak throughput increase of 10% between the two scenarios, its architecture appears to balance consistency and throughput effectively. It's important to understand what lead to ZooKeeper having no conflicts, it's lack of lock expiration. This design decision entails a significant tradeoff: while it avoids conflicts that typically arise from lease expiration, it also introduces potential vulnerabilities in scenarios involving client crashes. In such cases, locks held by clients with lingering connections persist until these connections timeout. This aspect of ZooKeeper's design is vital for understanding its behavior under conditions of partial system failures and the subsequent implications for system reliability and error recovery.

The introduction of network partitions and random sleeps revealed the critical importance of fencing tokens and lease mechanisms in preventing conflict scenarios and ensuring the integrity of locks. This emphasizes the need for distributed systems to incorporate robust error handling and recovery strategies to mitigate the risks associated with partitions and node failures.

8. Conclusion and Perspectives

This study underscores the complexity of achieving reliable distributed locking within HA key-value stores. It highlights that no one-size-fits-all solution exists; instead, the choice of a locking mechanism depends on the specific requirements of consistency, throughput, and fault tolerance of the application.

Redis offers ease of use and high throughput but requires careful consideration and additional mechanisms to ensure data consistency in distributed environments, specially when dealing with critical applications. Etcd provides strong consistency guarantees at the expense of throughput, making it suitable for applications where data integrity is critical. ZooKeeper presents a balanced option with good performance and reliability, suitable for a wide range of distributed locking scenarios.

Future research should explore the implications of newer technologies and algorithms, such as Redis-Raft, on the landscape of distributed locking. Additionally, developing standardized benchmarks and performance metrics will aid in more accurately assessing and comparing the effectiveness of distributed locking mechanisms across different key-value stores.

This study contributes to the broader understanding of distributed systems by providing insights into the performance and reliability of distributed locking mechanisms, offering valuable guidance for system designers and developers in choosing the appropriate technology for their specific needs.

Referências

Atlidakis, V., Andrus, J., Geambasu, R., Mitropoulos, D., and Nieh, J. (2016). Posix abstractions in modern operating systems: the old, the new, and the missing. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA. Association for Computing Machinery.

Authors, E. (2024). etcd. `https://github.com/etcd-io/etcd`.

D'amato, J. P., Dominguez, L., Stramana, F., Rubiales, A., and Perez, A. (2021). An hybrid cpu-gpu parallel multi-tracking framework for long-term video sequences. In Figueroa-García, J. C., Díaz-Gutierrez, Y., Gaona-García, E. E., and Orjuela-Cañón, A. D., editors, *Applied Computer Sciences in Engineering*, pages 263–274, Cham. Springer International Publishing.

Driscoll, K., Hall, B., Sivencrona, H., and Zumsteg, P. (2003). Byzantine fault tolerance, from theory to reality. In Anderson, S., Felici, M., and Littlewood, B., editors, *Computer Safety, Reliability, and Security*, pages 235–248, Berlin, Heidelberg. Springer Berlin Heidelberg.

Echavarría, S., Mejía-Gutiérrez, R., and Montoya, A. (2020). Development of an iot platform for monitoring electric vehicle behaviour. In Figueroa-García, J. C., Garay-Rairán, F. S., Hernández-Pérez, G. J., and Díaz-Gutierrez, Y., editors, *Applied Computer Sciences in Engineering*, pages 363–374, Cham. Springer International Publishing.

etcd (2024). etcd api guarantees.

Gray, C. and Cheriton, D. (1989). Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, page 202–210, New York, NY, USA. Association for Computing Machinery.

Grzesik, P. and Mrozek, D. (2019). Evaluation of key-value stores for distributed locking purposes. In Kozielski, S., Mrozek, D., Kasprowski, P., Małysiak-Mrozek, B., and Kostrzewa, D., editors, *Beyond Databases, Architectures and Structures. Paving the Road to Smart Data Processing and Analysis*, pages 70–81, Cham. Springer International Publishing.

Herlihy, M. P. and Wing, J. M. (1990). Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492.

Kamil, S. N. S., Thomas, N., and Elsanosi, I. (2021). Performance evaluation of zookeeper atomic broadcast protocol. In Zhao, Q. and Xia, L., editors, *Performance Evaluation Methodologies and Tools*, pages 56–71, Cham. Springer International Publishing.

Kingsbury, K. (2013a). Asynchronous replication with failover.

Kingsbury, K. (2013b). Jepsen: Zookeeper.

Kingsbury, K. (2020a). etcd 3.4.3.

Kingsbury, K. (2020b). Redis-raft 1b3fbf6.

Kleppmann, M. (2017). *Designing Data-Intensive Applications*. O'Reilly, Beijing.

Ongaro, D. and Ousterhout, J. (2014). In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA. USENIX Association.

Redis (2024). High availability with redis sentinel.

van Steen, M. and Tanenbaum, A. S. (2016). A brief introduction to distributed systems. *Computing*, 98(10):967–1009.

Xu, C. (2018). Research on data storage technology in cloud computing environment. *IOP Conference Series: Materials Science and Engineering*, 394(3):032074.

Yeo, C. S., Buyya, R., Pourreza, H., Eskicioglu, R., Graham, P., and Sommers, F. (2006). *Cluster Computing: High-Performance, High-Availability, and High-Throughput Processing on a Network of Computers*, pages 521–551. Springer US, Boston, MA.