Evaluation of Key-Value Stores for Distributed Locking Purposes

Piotr Grzesik(✉) and Dariusz Mrozek

Institute of Informatics, Silesian University of Technology, ul. Akademicka 16, 44-100 Gliwice, Poland
pj.grzesik@gmail.com, dariusz.mrozek@polsl.pl

Abstract. This paper presents the evaluation of key-value stores and corresponding algorithms with regard to the implementation of distributed locking mechanisms. Research focuses on the comparison between four types of key-value stores, etcd, Consul, Zookeeper, and Redis. For each selected store, the underlying implementation of locking mechanisms was described and evaluated with regard to satisfying safety, deadlock-free, and fault tolerance properties. For the purposes of performance testing, a small application supporting all of the key-value stores was developed. The application uses all of the selected solutions to perform computation while ensuring that a particular resource is locked during that operation. The aim of the conducted experiments was to evaluate selected solutions based on performance and properties that they hold, in the context of using them as a base for building a distributed locking system.

Keywords: Redis · Etcd · Consul · Zookeeper · Raft · Paxos · Zab · Redlock · Distributed computing · Cloud computing · Amazon Web Services · Python · Distributed locks

1 Introduction

The requirement of mutual exclusion in concurrent processing was identified over 50 years ago in paper [9] by Edsger Dijkstra. At present, most of the operating systems implement primitives that can be used to satisfy that requirement on a single machine, however, the dynamic growth of distributed computer systems in areas like artificial intelligence, data warehousing, and processing requires us to solve the locking problem in distributed systems in a reliable and fault-tolerant manner. Algorithms, like Paxos [22], Raft [25] or Zab [15] and their corresponding safety and liveness properties enabled and inspired implementation of consistent key-value stores like etcd [10], Consul [6] and Zookeeper [14], which can be used as a base for building distributed locking solutions; all of them offering primitives that make building such systems easier. Figure 1 shows the interaction between distributed locking service and two separate clients. Firstly, Client 1 acquires the lock and during a period of time when that lock is held, requests from Client

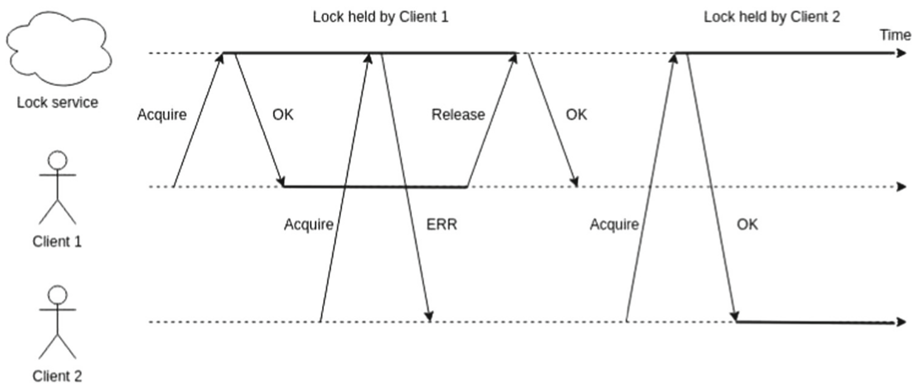

Fig. 1. Interaction with lock service by two clients

2 to acquire lock will fail. After the lock is released by Client 1, Client 2 can successfully acquire distributed lock.

When evaluating distributed locking mechanisms, depending on a use case and correctness requirements, it is very important to determine, if the mechanism satisfies some or all of the following properties:

- Safety – property that satisfies the requirement of mutual exclusion, ensuring that only one client can acquire and hold the lock at any given moment.
- Deadlock-free – property that ensures that, eventually, it will be possible to acquire and hold a lock, even if the client currently holding the lock becomes unavailable.
- Fault tolerance – property ensuring that as long as the majority of the nodes of the underlying distributed locking mechanism are available, it is possible to acquire, hold, and release locks.

Deadlock-free and fault tolerance can be also summed up as one, liveness property.

The aim of the research presented in the paper is to evaluate and compare key-value stores with corresponding algorithms in regard to available mechanisms that can be used for performing distributed locking. For the purposes of testing, four solutions were selected, etcd, Redis [26] with Redlock [28], Consul and Zookeeper. Selection of the key value stores was made based on the properties of their underlying algorithms like Raft and Zab as well as their popularity and widespread adoption. All proposed services were evaluated based on performance, their safety, deadlock-free and fault tolerance properties as well as on ease-of-use both in client applications and in setting up infrastructure needed for them to run in a fault tolerant manner. As a base for research, a small Python 3.7 application was developed, that is able to perform distributed locking with all of the mentioned solutions.

2 Related Works

In the literature, there is few research concerning the comparison of key-value stores performance. In the research [23], Gyu-Ho Lee is comparing key write performance of Zookeeper, etcd, and Consul working in three-node clusters. The author is measuring performance with regards to disk bandwidth, network traffic, CPU utilization as well as memory usage. The author concludes the research with claims that etcd offers better throughput and average latency while using less memory than other solutions. However, it is also noted that Zookeeper offers the lowest minimal latency, at the cost of potential higher average latency.

Patrick Hunt in his analysis [12] evaluates Zookeeper server latency under varying loads and configurations, which included operations like creating permanent znodes[1], setting, getting and deleting znodes, as well as creating ephemeral znodes and performing znodes watches. He observed that for a standalone server, additional cores (testing was performed with 1, 2 and 4 CPUs) did not provide significant performance gains. He also observed that in general, Zookeeper is able to handle more operations per second with a higher number of clients (around 4 times more operations per second between 1 and 10 clients).

Redis documentation [27] offers a detailed performance tests. It also highlights factors that are critical for Redis performance, such as significantly worse performance while running on a virtual machine compared to running without virtualization on the same set of hardware. It also notes that due to single-threaded nature of Redis, it performs much better on CPUs with larger caches and fewer cores, such as Intel Sandy Bridge CPUs, which perform up to 50% better in comparison to corresponding AMD Opteron CPUs.

The liveness and safety properties of Raft were presented in paper [25] by Diego Ongaro, where the same properties of Zab, the algorithm underpinning Zookeeper, were presented [15] by Junqueira, Reed and Serafini. Redlock was developed and described by Salvatore Sanfilippo, however safety property was later disputed by Martin Kleppmann, which presented in his article [19], that under certain conditions, the safety property of Redlock might not hold, suggesting that Redlock might not be the most optimal solution where correctness is the main objective.

In his paper [4], Mike Burrows from Google described an architecture for Chubby, lock service for distributed systems. One of the important decisions made by the team developing Chubby was to create a separate service, instead of a library, motivating that decision by the easier implementation for clients (in comparison to integrating consensus protocol into the applications). Chubby uses Paxos as an underlying consensus mechanism, ensuring safety property.

Kyle Kingsbury in his works [17,18] evaluated etcd, Consul and Zookeeper with Jepsen [13], the framework for distributed system verification. During verification, it was revealed that Zookeeper is able to preserve linearizability in the presence of network partition and leader election. The same did not hold true for

[1] Zookeeper uses a hierarchical namespace, where every node is called znode.

Consul and etcd, which experienced stale reads[2]. This research prompted maintainers of both etcd and Consul to provide mechanisms that enforce consistent reads.

While there are a few research works focusing on certain aspects like performance or correctness of the selected solutions, none of them provides a detailed comparison of them in the context of using them as a building block of a distributed locking system.

3 Key-Value Stores

Key-value stores, also commonly called key-value databases, are certain type of NoSQL [29] databases, which employ, unlike established SQL databases, schemaless data model. As the name suggests, stored data is represented in form of key-value pairs, where values can be arbitrary binary objects, which makes it the most flexible data store from application perspective. Thanks to the simple structure, key-value stores like Redis can offer very high performance in comparison to traditional SQL databases as well as other types of NoSQL databases [16]. For purposes of this research, the most interesting key-value stores are those that offer strict data consistency and high availability (falling into CP of CAP theorem presented by Brewer [3]) which in combination with performance, can serve as a solid base for building fast and responsive distributed locking mechanisms. Out of the existing stores, four were selected: etcd, Consul, Zookeeper, and Redis, based on their wide-spread usage (in Hadoop ecosystem [11], Kubernetes [20], Nomad [24]), properties of underlying consensus algorithms and ease-of-use.

3.1 Etcd

Etcd is an open source, distributed key-value store written in Go, currently developed under Cloud Native Computing Foundation [5]. It uses the Raft consensus algorithm for management of highly available replicated log, being able to tolerate node failures, including the failure of the leader. In addition, it offers dynamic cluster membership reconfiguration. Etcd enables distributed coordination by implementing primitives for distributed locking, write barriers and leader election. It uses persistent, multi-version, concurrency-control data model. Client libraries in languages like Go, Python, Java, and others are available, as well as command-line client "etcdctl".

3.2 Consul

Consul is an open source, distributed key-value store written in Go, developed by Hashicorp. In addition to being a consistent key-value store, Consul can be used for health checking, service discovery or as a source of TLS certificates for

[2] Stale read is a read operation that fetches result which does not reflect all updates to the given value.

providing secure connections between services in a system. It also has support for multiple data centers, with a separate Consul cluster in each data center. Similarly to etcd, it uses the Raft consensus algorithm for managing replicated log and also can be used for distributed coordination with sessions mechanism enabling distributed locks. In addition, Consul offers several consistency modes for reads, depending on the application needs. Consul nodes can be either servers or clients with only servers taking part in Raft consensus protocol. Clients use the Gossip protocol to communicate with each other.

3.3 Zookeeper

Zookeeper is an open source, highly available coordination system, written in Java. Initially developed at Yahoo, currently is a project maintained by Apache Software Foundation [2]. Zookeeper uses the Zab atomic broadcast protocol. Nodes store data in a form of a hierarchical namespace, resembling file system, where each node in Zookeeper's tree is called a znode. Zookeeper's main use cases are leader election, group membership, a configuration store, distributed locking and priority queues. While not being strictly a key-value store, it can be also used and qualified as such for purposes of this research. Zookeeper's Java client library, Apache Curator, offers high-level API with implemented recipes for elections, locks, barriers, counters, caches, and queues. Similar recipes are also available as a part of the Python client library, kazoo.

3.4 Redis

Redis is an open source, in-memory key-value database, that offers optional durability. It was developed by Salvatore Sanfilippo and is written in ANSI C. Redis offers support for data structures such as lists, sets, sorted sets, maps, strings, hyperloglogs, bitmaps, and streams. Redis exhibits very high performance in comparison to other database offerings [16]. According to the DB-Engines ranking [8], it is the most popular key-value database. While mostly used for caching, queuing and Pub/Sub, it can also be used, in combination with the Redlock, as a base for building distributed locking mechanism.

4 Environment and Implementation

For the purposes of testing, a small Python 3.7 application was developed, that can use each of the stores presented in the previous chapter to acquire distributed lock, simulate a short computation that requires the lock to be held, and release the lock afterwards. In addition, it measures the time taken to acquire a distributed lock. Consul, etcd, and Zookeeper were tested as 3-node clusters, using configuration presented on Fig. 2. The whole needed infrastructure was deployed in Amazon Web Services cloud offering as EC2 instances, each of which has been provisioned in different availability zone in the same geographical region, eu-central-1, to ensure fault tolerance of a single availability zone., while maintaining latency between cluster instances in sub-milliseconds range. All instances (including the client node) are of type “t2.medium”, which have the following specification:

- OS - Amazon Linux 2 [1]
- AMI[3] ID - ami-0cfbf4f6db41068ac
- CPU - 2 vCPUs of 3.3 GHz Intel Scalable Processor
- 4 GiB of RAM
- 8 GiB of EBS[4] SSD storage

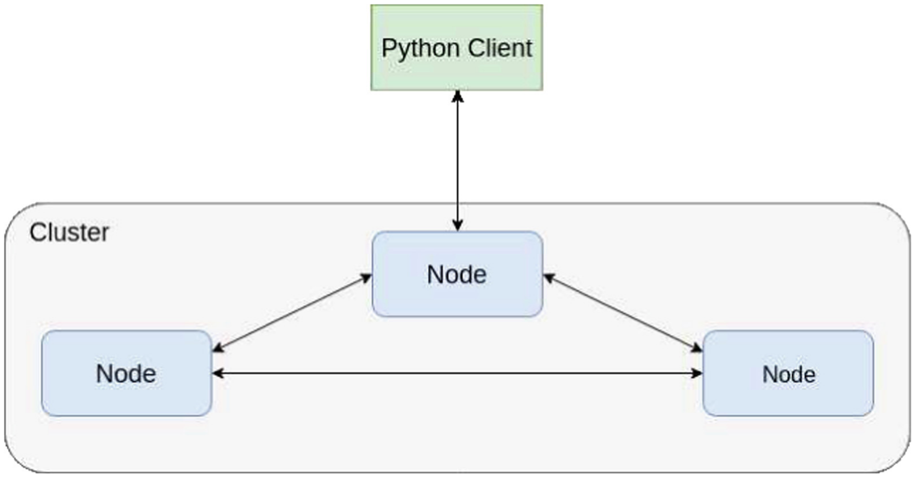

Fig. 2. Environment configuration diagram for etcd, Consul and Zookeeper

Redis has been deployed in a similar environment, however the usage of the Redlock locking algorithm required to use 3 standalone Redis nodes instead of having 3-nodes cluster. Configuration is presented on Fig. 3.

4.1 Etcd Implementation and Configuration

To build an etcd cluster, the etcd in version 3.3.9 was used, compiled with Go language with version 1.10.3. The cluster was configured to ensure sequential consistency model [21], which satisfies the safety property of the solution. To communicate with the cluster, the client application used the python-etcd3 library, which implements locks using the atomic compare-and-swap mechanism,

[3] Amazon Machine Images, image that is used to create virtual machines using Amazon Elastic Compute Cloud.
[4] Elastic Block Storage, persistent block storage offering from Amazon Web Services.

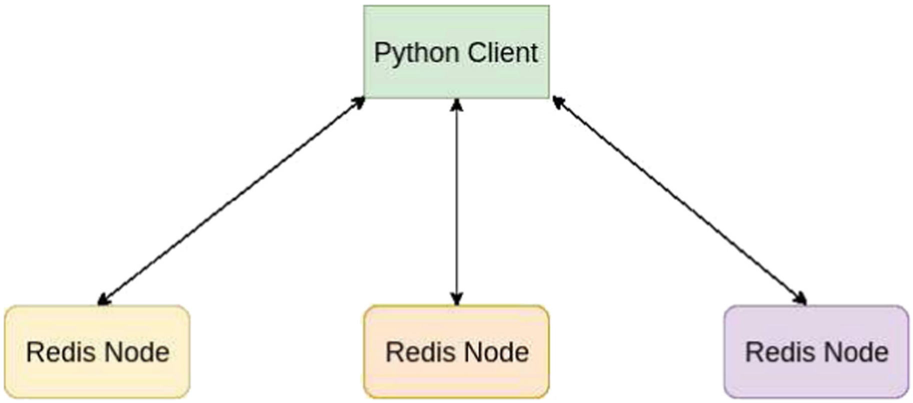

Fig. 3. Environment configuration diagram for Redis

that checks if the given key is already set and if not, atomically sets it to a given value, thus acquiring the lock. If the key is already set, that means the lock is already acquired. To ensure deadlock-free property, the TTL (Time-To-Live) for a lock is set, which releases the lock, if the lock-holding client crashes or is network partitioned away. Thanks to the underlying Raft consensus algorithm, fault tolerance property is also satisfied. It is worth noting that etcd from version 3.2.0 supports native Lock and Unlock RPC methods, however, python-etcd3 as of yet does not support those methods.

4.2 Consul Implementation

Consul cluster was assembled using version 1.4.0, compiled with Go 1.11.1 and configured with "consistent" consistency mode [7], ensuring the safety of the underlying Raft protocol. Usage of the Raft consensus protocol also ensures that fault tolerance property is satisfied. Application acquires locks by using the consul-lock Python library, which implements locks using sessions mechanism with the check-and-set operation. According to the Consul documentation [6], depending on the selected health checking mechanism, either safety or liveness property may be sacrificed. The selected implementation uses only session TTL health check, which guarantees that both properties hold, however, it's worth noting that TTL is applied at the session level, not on specific lock and session timeout either deletes or releases all locks related to the session.

4.3 Zookeeper Implementation

Zookeeper cluster was build using version 3.4.13, running on Java 1.7, with OpenJDK Runtime Environment. Thanks to the Zab protocol properties, safety and fault tolerance requirements are satisfied. To implement application interacting with the Zookeeper cluster, the Python kazoo library with lock recipe was used.

This implementation takes advantage of Zookeeper's ephemeral and sequential znodes. On lock acquire, a client creates znode with ephemeral and sequential flag and checks, if sequence number of that znode is the lowest, and if that is true, the lock is acquired. In opposite situation, client can watch the znode and be notified when the lock is released so it can once again try to acquire the lock. Thanks to usage of ephemeral znodes, we can achieve deadlock-free property, because if the client holding lock becomes unavailable, the ephemeral node will be destroyed resulting in release of the lock.

4.4 Redis with Redlock Implementation

The last solution is build on top of 3 standalone Redis nodes with version 5.0.3. The application implementing locking mechanisms is using the Redlock algorithm, provided by the Python redlock library. The Redlock algorithm works by sequentially trying to acquire lock on all independent Redis instances and the lock is acquired when it was successfully acquired on majority of the nodes. Specific details of Redlock algorithm are presented in [28]. Thanks to the requirement to lock on the majority of the nodes before considering the lock to be acquired, safety property is satisfied. Similarly to the other solutions, the deadlock-free property is satisfied based on lock timeouts. Fault tolerance is satisfied as well, by tolerating up to (N/2) - 1 failures of the independent Redis nodes. It is worth noting, that under specific circumstances, described by Martin Kleppmann in his article [19], it is possible for the Redlock to lose the safety property, making it not suitable for applications in which correctness cannot be sacrificed under any circumstances.

4.5 Implementations Summary

In this chapter, we described the implementation and configuration details related to all selected solutions. In cases of Zookeeper, etcd and Consul, the safety, deadlock-free and fault tolerance properties are satisfied thanks to selected cluster configurations, their mechanisms like sessions for Consul and ephemeral nodes for Zookeeper, as well as their underlying algorithms like Raft and Zab. In the case of Redis, the deadlock-free property is satisfied thanks to the selected Redlock algorithm and the size of the cluster used for experiments allows for failure of one node, hence satisfying the fault tolerance property. The safety property for Redlock is not always satisfied, which was described by Martin Kleppmann in his analysis [19]. In the next chapter, we focused on the performance evaluation of all selected solutions.

5 Performance Experiments

To evaluate the performance of the selected solutions, we performed several experiments. Firstly, we conducted the experiment to evaluate the behavior of all solutions for a workload where multiple processes have to acquire distributed locks on various resources, but where we do not experience the clients competing for a particular lock. In this step, for each service, we simulated concurrent traffic from 1, 3, and 5 different processes that are trying to acquire locks on different lock keys, with each simulation lasting 2 min. Secondly, we ran a simulation to evaluate the behavior of selected solutions for a workload, where we experience high contention over a particular lock key, with multiple processes trying to acquire the same lock simultaneously. In this step, once again, we simulated concurrent traffic from 1, 3, and 5 different client applications, but this time, all of them tried to acquire the same lock key. Based on the results of the simulations, we examined metrics related to lock acquire time for each configuration and evaluated all of the solutions with regard to expected workload patterns and requirements.

5.1 Acquiring Different Lock Keys

In the first simulation, we evaluated the locking of different lock keys. Obtained results are presented in Table 1. For the etcd, we observe that with an increasing number of client processes, the average lock acquire time is also increasing, which is similar for Consul and Zookeeper as well. For Redis, we see very small changes in the average lock time, with the average time for 5 processes being even lower than for 3 processes. Redis also has the lowest average lock acquire time from all of the tested solutions, with 1.3 ms for 5 processes, Zookeeper is next with 5.9 ms after that is etcd with 7.68 ms and Consul at the end with 16.52 ms. Maximum lock acquires time, the 90th percentile of lock acquire time, and the 99th percentile of lock acquire time were also tested, to evaluate the stability of expected lock acquire time. In all tested solutions with 5 processes, the 90th percentile was about 20% higher than average time, however, while for etcd, Consul and Zookeeper 99th percentile was about 2 times higher than the average, for Redis we observed 4 times higher value of the 99th percentile in comparison to the average time. However, even with that change, the 99th percentile of lock acquire time of Redis was still almost 2 times smaller than those of other tested solutions.

Table 1. Summary of results for different keys simulations

	etcd			Consul			Zookeeper			Redis		
No. of proc.	1	3	5	1	3	5	1	3	5	1	3	5
Avg (ms)	4.91	6.8	7.68	11.7	13.6	16.52	3.64	4.53	5.9	1.2	1.36	1.3
Median (ms)	4.66	6.49	7.33	11.5	13.12	16.01	3.41	4.4	5.74	1.12	1.26	1.25
Min (ms)	3.99	4.43	4.19	10.5	10.42	10.38	2.86	2.79	2.76	0.95	0.97	0.98
Max (ms)	20.97	21.75	28.43	28.3	45.38	119.71	36.64	15.35	19.98	12.82	14.13	9.42
90p (ms)	5.51	8.32	9.31	12.42	16.09	20.29	4.3	5.97	7.4	1.32	1.56	1.49
99p (ms)	9.38	13.41	16.1	15.09	19.74	25.07	7.49	7.65	9.97	2.93	3.37	5.07

5.2 Contesting over the Same Lock Keys

In the second simulation, we tested the behavior of locking applications that were contesting over the same lock key. Results gathered during the simulation are presented in Table 2. While inspecting the average lock acquire time, we observed a general trend of increasing time with more client processes for all tested solutions. It is, however, worth noting, that while for etcd, Consul and Zookeper changes in average lock time between 1, 3 and 5 processes are relatively small, for Redis we observe 3 times increase while changing from 1 to 3 processes and almost 5 times increase for 5 processes. Even with that behavior, Redis still has the lowest average lock time among all tested solutions, but only 0.24 ms lower (for 5 concurrent processes) than Zookeeper. Results of the maximum, the 90th percentile and the 99th percentile of the lock acquire time reveal that Redis-based solution is prone to having instances of very high lock acquire times. While the 90th percentile is very small (1.32 ms), the 99th percentile is over 200 ms with maximum lock acquire time being 411.2 ms. All other solutions do not exhibit such anomalies, all having the 99th percentile for 5 processes roughly 2 times higher than average lock acquire time. The best performing solution based on that metric is Zookeeper, with 7.49 ms and 10.17 ms of the 99th percentile for 3 and 5 processes, respectively.

Table 2. Summary of results for the same keys simulations

	etcd			Consul			Zookeeper			Redis		
No. of proc.	1	3	5	1	3	5	1	3	5	1	3	5
Avg (ms)	4.91	6.98	7.6	11.7	13.84	16.36	3.64	4.82	5.94	1.2	3.78	5.7
Median (ms)	4.66	6.48	7.22	11.5	13.49	15.89	3.41	4.65	5.79	1.12	1.15	1.1
Min (ms)	3.99	4.23	4.16	10.5	10.62	10.18	2.86	2.84	3.07	0.95	0.96	0.93
Max (ms)	20.97	38.7	47.13	28.3	151.96	95.05	36.64	20.93	19.98	12.82	414.52	411.2
90p (ms)	5.51	8.5	9.16	12.42	16	20.85	4.3	6.3	7.4	1.32	1.3	1.3
99p (ms)	9.38	14.14	16.79	15.09	20.1	26.25	7.49	8.9	10.17	2.93	5.07	203.87

5.3 Results Summary

Comparing results from both simulations, we can observe that for the etcd, Zookeeper and Consul there is little difference in performance between situations where client processes are contesting over the same key or are concurrently acquiring different lock keys. The same does not hold for Redis, which exhibits stable and the best performance of all tested solutions in case of clients accessing different lock keys, while being unstable with workload involving multiple clients contesting over the same key, having the 99th percentile for 5 processes as high as 203.87 ms, over 10 times more than other tested solutions. Considering both workloads, the most performing solution is Zookeeper, which in worst case scenario (5 processes, different lock keys) offers average lock time of 5.94 ms, with the maximum of 19.98 ms, the 90th percentile of 7.4 and the 99th percentile of 10.17 ms. It is worth noting that if upper lock time bound is not essential for the given application, then the best performing solution is Redis, which for the same workload offers the 90th percentile lock acquire time of 1.3 ms.

6 Concluding Remarks

The aim of this research was to implement and compare distributed locking applications built on top of selected key-values stores. To evaluate the performance of selected solutions, an application supporting distributed locking was developed. We believe that experiments and analysis of the results presented in this paper complement performance comparisons presented in [23], as well as analysis of safety and correctness by Kyle Kingsbury [17,18] and Martin Kleppmann [19] to serve as a comprehensive guide to selecting the base for building distributed locking systems.

All solutions were also evaluated based on ease-of-use both from the infrastructure and implementation perspective. When it comes to setting up infrastructure, Redis with Redlock is the easiest to setup, because all nodes are independent and can be easily replaced. For Zookeeper, Consul and etcd, it is slightly more complicated, because all nodes have to form a cluster communicating with each other. Zookeeper, prior to version 3.5.0 also does not offer cluster membership changes without restarting nodes. All solutions have easy to use Python client libraries, however, Zookeeper's kazoo with corresponding recipes is the most mature and robust client library.

The second axis on which the comparison was performed is related to safety, deadlock-free and fault tolerance properties. Zookeeper, etcd and Consul with correct configurations satisfy all considered properties, making them suitable for applications for which those properties are a strict requirement. Redis with Redlock satisfy both deadlock-free and fault tolerance properties, however, under some circumstances, safety property might be sacrificed, which makes it a questionable choice for cases where safety and correctness is a top priority.

The third and final comparison of all solutions was based on metrics related to lock acquire time. It was observed that for workload patterns where we do not observe high contention over the same lock key, Redis is the best performing solution, however with the growing number of concurrent processes trying to acquire the same lock key, Redis performance deteriorates and we observe cases in which worst-case lock acquire time is 10 to 20 times higher than other solutions. For workloads with such patterns and for which worst-case scenario performance is critical, Zookeeper emerged as the most suitable solution.

References

1. Amazon Linux 2. https://aws.amazon.com/amazon-linux-2/. Accessed 9 Jan 2019
2. Apache Software Foundation. https://www.apache.org/. Accessed 11 Jan 2019
3. Brewer, E.: CAP twelve years later: how the "Rules" have changed. Computer **45**, 23–29 (2012). https://doi.org/10.1109/MC.2012.37

4. Burrows, M.: The Chubby lock service for loosely-coupled distributed systems. In: 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI) (2006)
5. Cloud Native Computing Foundation. https://www.cncf.io/. Accessed 11 Jan 2019
6. Consul Documentation. https://www.consul.io/docs/index.html. Accessed 11 Jan 2019
7. Consul consensus protocol. https://www.consul.io/docs/internals/consensus.html. Accessed 11 Jan 2019
8. Corporation, N.: DB engines ranking. https://db-engines.com/en/ranking
9. Dijkstra, E.: Solution of a problem in concurrent programming control. Commun. ACM **8**(9), 569 (1965)
10. etcd Documentation. https://etcd.readthedocs.io/en/latest/. Accessed 10 Jan 2019
11. Hadoop Documentation. https://hadoop.apache.org/. Accessed 11 Jan 2019
12. Hunt, P.: Zookeeper service latencies under various loads and configurations. https://cwiki.apache.org/confluence/display/ZOOKEEPER/ServiceLatencyOverview. Accessed 9 Jan 2019
13. Jepsen - Distributed Systems Safety Research. https://jepsen.io/. Accessed 9 Jan 2019
14. Junqueira, F., Reed, B., Hunt, P., Konar, M.: ZooKeeper: wait-free coordination for Internet-scale systems. In: Proceedings of the 2010 USENIX conference on USENIX Annual Technical Conference, June 2010
15. Junqueira, F., Reed, B., Serafini, M.: Zab: high-performance broadcast for primary-backup systems. In: IEEE/IFIP 41st International Conference on Dependable Systems and Networks (DSN), pp. 245–256, June 2011
16. Kabakus, A.T., Kara, R.: A performance evaluation of in-memory databases. J. King Saud Univ. Comput. Inf. Sci. **29**(4), 520–525 (2016)
17. Kingsbury, K.: Jepsen: etcd and Consul. https://aphyr.com/posts/316-call-me-maybe-etcd-and-consul. Accessed 9 Jan 2019
18. Kingsbury, K.: Jepsen: Zookeeper. https://aphyr.com/posts/291-call-me-maybe-zookeeper. Accessed 9 Jan 2019
19. Kleppmann, M.: How to do distributed locking. http://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html. Accessed 9 Jan 2019
20. Kubernetes Documentation. https://kubernetes.io/. Accessed 11 Jan 2019
21. KV API Guarantees. https://coreos.com/etcd/docs/latest/learning/api_guarantees.html. Accessed 11 Jan 2019
22. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998)
23. Lee, G.H.: Exploring performance of etcd, zookeeper and consul consistent key-value datastores. https://coreos.com/blog/performance-of-etcd.html. Accessed 9 Jan 2019
24. Nomad Documentation. https://www.nomadproject.io/. Accessed 11 Jan 2019
25. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference, pp. 305–320, June 2014
26. Redis Documentation. https://redis.io/documentation. Accessed 11 Jan 2019
27. How fast is Redis? https://redis.io/topics/benchmarks. Accessed 9 Jan 2019
28. Distributed locks with Redis. https://redis.io/topics/distlock. Accessed 11 Jan 2019
29. Sullivan, D.: NoSQL for Mere Mortals, 1st edn. Addison-Wesley Professional, Boston (2015)