Volume 1, 2025, Pages 32-40 *Elaris Computing Nexus*
| Regular Article | Open Access

Evaluating Performance and Fault Tolerance of a Linearizable Distributed Cache in Comparison to Redis Cluster

Fengbin Sun
University of Science and Technology of China, Baohe District, Hefei, Anhui, China, 230052.
febsun@ustc.edu.cn

Article Info
Elaris Computing Nexus
https://elarispublications.com/journals/ecn/ecn_home.html

https://doi.org/10. XXXX/ECN/2025004

Received 10 February 2025
Revised from 18 March 2025
Accepted 20 April 2025
Available online 30 April 2025
Published by Elaris Publications.

Corresponding author(s):
Fengbin Sun, University of Science and Technology of China, Baohe District, Hefei, Anhui, China, 230052.
Email: febsun@ustc.edu.cn

Abstract – We present the experimental findings of a proposed linearizable cache system, which is subsequently compared to those of Redis Cluster, a widely utilized distributed caching solution. Both were implemented using Docker to establish a distributed environment with failure management. The evaluation encompassed measuring read and write requests, analyzing latency percentiles (P25, P50, P80, P90, P99), and assessing fault tolerance concerning disk failure, network delay, network partition, and node failure. The results indicate that the linearizable cache outperforms Redis Cluster in terms of read latency but incurs higher write latency due to strong consistency. It maintains consistency during a majority node failure, unlike Redis Cluster, which leads to data loss and inconsistencies. The scalability seemed promising with 3, 5, and 7 nodes; nevertheless, a storage constraint was encountered in the proposed system as the cluster size expanded.

Keywords – Linearizable Cache, Redis Cluster, Distributed Systems, Fault Tolerance, Performance Evaluation, Docker, Scalability, Consistency Guarantees, Latency, Benchmarking.

I. INTRODUCTION

Distributed caching has become an essential element in contemporary distributed systems architecture, functioning as a crucial method for improving application performance, minimizing network latency, and decreasing database load. As distributed systems grow in complexity and size, the efficacy of caching techniques has become increasingly crucial in influencing overall system performance. Alubady, Salman, and Mohamed [1] assert that caching solutions in distributed systems necessitate distinct attention beyond conventional single-node implementations due to issues with data consistency, network segmentation, and resource allocation in varied contexts. **Fig. 1** depicts the mechanism of distributed caching. It illustrates the distribution of tasks into distinct nodes, from applications to databases, facilitating operational efficiency for companies.

Distributed caching algorithms must reconcile several conflicting objectives: optimizing hit ratios, reducing latency, maintaining data consistency, and effectively managing memory resources across dispersed nodes. The choice of suitable caching algorithms is significantly influenced by workload factors, system design, and unique application needs. Salah et al. [2] observed that the rise of microservices and serverless architectures has hampered the deployment of caching strategies, requiring more advanced methods for distributed cache management.

In distributed systems, consistency models define standards for data synchronization and dictate how users and programs should interpret data modifications across several nodes. In a distributed system, it explicitly regulates data access and modification across several nodes and how clients are notified of these updates. These models encompass both stringent and lenient methodologies. Consistency models in distributed systems manifest in numerous varieties. Each consistency model possesses distinct advantages and disadvantages, and the specific requirements of the system will dictate the most suitable model.

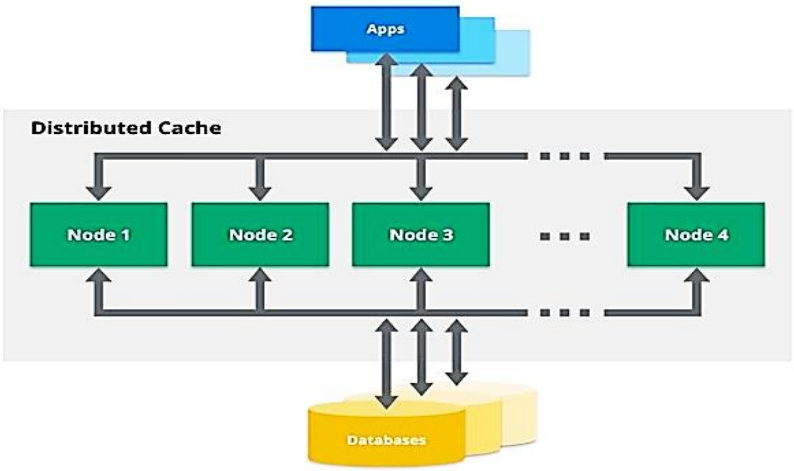

Fig 1. Distributed Caching

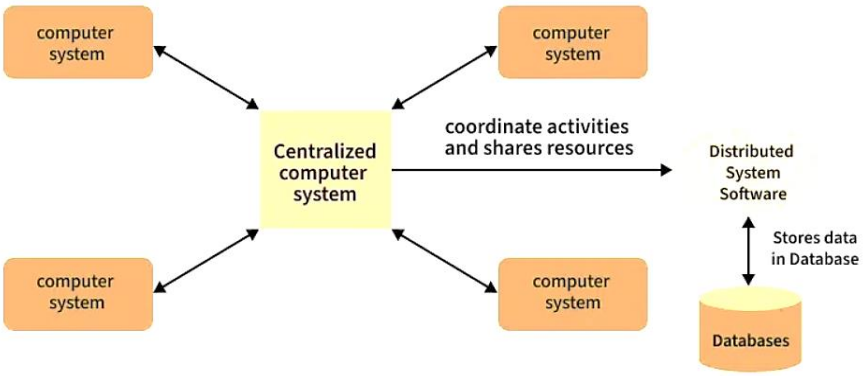

Fig 2. Consistency Model in Distributed System

Consistency models facilitate the equilibrium of data precision, velocity, and accessibility in distributed systems. Strong Consistency guarantees current data, whereas Weak Consistency emphasizes speed. Causal and Sequential Consistency maintain the order of actions, which is beneficial for collaborative applications. Session Consistency maintains data stability within sessions, whereas Monotonic Reads/Writes ensure predictable interactions. Selecting the appropriate model improves system efficiency and user satisfaction (see **Fig. 2**).

Distributed systems consist of networked nodes that have a common objective in their operations. To address a particular issue, the task has been segmented into minor assignments assigned to those devices. Each node performs its designated work and transmits the findings to the submission device. Moreover, decentralized frameworks can be heterogeneous (P2P, Cloud, and Grid) and homogeneous (clusters), and they face several challenges such as fault tolerance, load balancing, resource selection, and QoS (quality of services).

Fault tolerance examines a system's response to unexpected software or hardware failures, particularly in contrast to uniprocessors; detecting failures in distributed systems is challenging. Fault tolerance fundamentally comprises two essential components: failure detection and recovery. Maintaining system functionality after failures or when components are disconnected or defective poses a significant difficulty in distributed systems. Numerous fault tolerance strategies exist within the distributed paradigm, including retry, replication, checkpointing, and message logging, among others. We focus on evaluating the performance and fault tolerance of a proposed linearizable cache system in comparison with Redis Cluster within a distributed environment. Specifically, we examine the systems' behavior under various failure scenarios, such as disk failures, network delays, and node failures, while also testing their scalability with different cluster sizes.

The remainder of this work is organized as follows: Section II summarizes the distributed cache architectures relevant to our study. These include client-server architectures, P2P architecture, and hierarchical architectures. Section III reviewed related work focused on DCS optimization for performance enhancement, impact of consistency models, DCS benchmarking, and Redis in DCS. Section IV describes our experimental setup, which highlights our system configuration and environment, benchmarking metrics, and failure simulation. Section V provides a detailed discussion of the findings we obtained in our study. Section VI concludes the study and highlights the tradeoffs between fault tolerance, performance, and consistency within distributed caching systems.

II. DISTRIBUTED CACHE ARCHITECTURES

Client-Server Architecture

The client-server system has gained immense popularity in contemporary computing due to its ubiquitous use across several platforms. The established protocols utilized for communication between clients and servers include: HTTP (hypertext transfer protocol), SMTP (simple mail transfer protocol), and FTP (file transfer protocol). A client-server model is defined as a software infrastructure consisting of both a server and the clients, where clients consistently issue requests and the server answers to these requests. The model facilitates inter-procedure communication through data exchange between the server and client, each executing distinct functions (see **Fig. 3**).

The model serves as the fundamental model for distributed caching systems. In this configuration, cache clients engage with a specialized cache cluster over established protocols. The cache client employs a hashing or sharding method to identify the suitable node or partition for data storage while caching information. This design delineates distinct responsibilities, with clients tasked with cache key extraction and routing, while the cache cluster oversees data storage, replication, and consistency. The client library generally includes advanced routing logic that identifies the suitable cache node for every operation according to the established sharding algorithm.

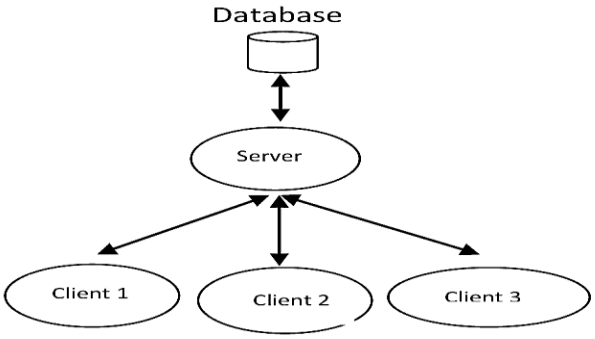

Fig 3. Inter-Procedure Communication Between Server and Client

P2P Architecture

A quick survey of the literature reveals a substantial array of meanings for "peer-to-peer," mostly differentiated by the degree of breadth ascribed to the phrase. The most stringent definitions of "pure" Peer-to-Peer (P2P) pertain to entirely dispersed systems, when all nodes are entirely similar regarding functionality and the tasks they do. These definitions do not include, for instance, systems utilizing the concept of "supernodes" (nodes that serve as dynamically designated localized mini-servers), such as Wilson [3], which are broadly recognized as P2P, nor do they account for systems that depend on a centralized server infrastructure for certain non-essential functions (e.g., bootstrapping, reputation management, etc.).

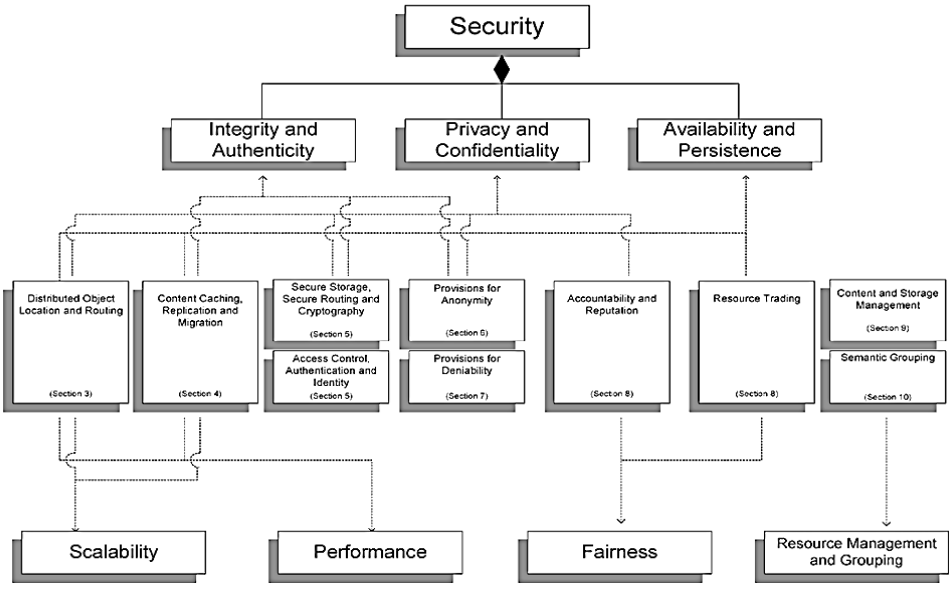

Fig 4. Design Factors Influencing the Primary Characteristics of P2P Distributed Content Networks

Androutsellis-Theotokis and Spinellis [4] provides a comprehensive and widely recognized definition of "P2P" as a category of applications that utilize resources—such as storage, processing power, content, and human presence—located at the periphery of the internet. This definition, however, includes systems that entirely depend on centralized servers for their functionality (such as seti@home, several instant messaging platforms, or the infamous Napster), along with other applications from the domain of Grid computing. The scholars compiled the diverse design features and solutions, along

with their correlation to pertinent nonfunctional characteristics, through a comprehensive analysis of existing, researched, or proposed P2P content distribution systems (see **Fig. 4**).

P2P distributed caching designs abolish the differentiation between clients and servers, enabling each node to store cached data and fulfill requests from other nodes. This method provides intrinsic scalability benefits and eliminates single points of failure, although it complicates consistency control and cache coherence.

Hierarchical Architectures

Hierarchical caching designs structural cache nodes across various layers, generally featuring local caches at the periphery nearest to applications, regional caches supporting several local caches, and global caches serving as the primary data source. This method enhances both performance and network effectiveness by positioning frequently accessed data nearer to consumers.

Hierarchical caching is already a prevalent reality in much of the Internet. Numerous ISPs and Internet-connected institutions have been implementing caches to minimize bandwidth consumption and reduce latency for their consumers. When a document is solicited across a caching hierarchy: i) each level of the hierarchy incurs supplementary delays, ii) upper-level caches may become bottlenecks and experience prolonged queuing delays, and iii) several copies of the document are retained at various cache levels. Conversely, distributed caching eliminates intermediary caches that cause extra delays. Documents are solicited directly from institutional caches, which often see minimal load. The majority of traffic traverses lower network tiers, which experience reduced congestion. No supplementary copies are retained in intermediate caches, conserving disk space.

Cache management in hierarchical cache designs continues to be an unresolved challenge. Current studies generally address local replica management within a singular cache layer via cache replacement strategies. In hierarchical cache designs, the distinct read/write attributes of different storage media introduce significant issues for replica management. To circumvent the line-speed constraints of ICN in-network caching while offering terabyte-scale single-node cache capacity, Huo et al. [5] implement a dual-tier cache architecture that integrates DRAM and SSD.

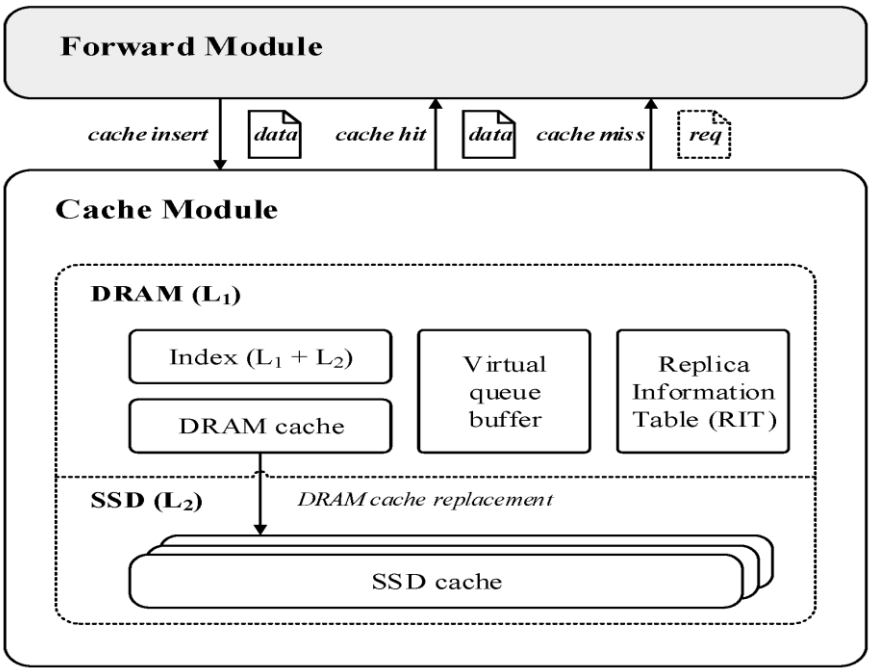

Fig 5. Depiction of Hierarchical Cache Design

As illustrated in **Fig. 5**, we adopt the split architecture design, partitioning each ICN cache router into two functional components: a forwarding module and a caching module. This method resolves the problem of SSDs slow I/O obstructing the forwarding pipeline by divorcing packet forwarding from caching activities in terms of timing.

III. RELATED WORK

DCS and Algorithm Optimization for Performance Improvement

Kowarschik and Weiß [6] propose and analyze novel algorithms or cache hierarchies, comparing their performance to baseline methods, with the primary aim of minimizing CPU and disk consumption at various levels. Distributed systems have emerged as a focal point for protein folding, offering high-performance computing capabilities for researchers addressing larger and more complex issue situations.

Nevertheless, time-consuming problem cases are not readily resolved through parallelism (or at the expected ratio of time compared to a serial processor) due to many overhead considerations that substantially impair performance. Given that

these elements render the system environment distinct from uni-processor systems, it is imperative to create and implement more appropriate algorithms.

Meng et al. [7] delineated a distributed caching strategy (DCS) that enhances a caching policy for distributed systems. During a cache migration, each processor is engineered to extract data from its local cache and transmit it to the adjacent processor. Each CPU acquires data from its neighboring processor and saves the information in its local cache. To obtain a local cache, we employ a modified beam search method for its efficacy and efficiency.

Impact of Consistency Models and Fault Tolerance on DCS

As described by Borst, Gupta, and Walid [8], the selection of a consistency model profoundly impacts the implementation and performance of distributed caching algorithms. Robust consistency models necessitate significant coordination across nodes, potentially compromising the latency advantages of caching. They investigated the performance effects of various consistency models on distributed caching, revealing that eventual consistency systems attained throughput rates 3-5 times greater than those of strongly consistent models. Nonetheless, this performance benefit must be considered in relation to application-specific consistency demands. Numerous contemporary distributed caching systems utilize hybrid consistency models, enabling applications to define consistency criteria for each request individually. This method facilitates precise modification of the consistency-performance tradeoff according to the significance of certain data.

As decribed by Petrot, Greiner, and Gomez [9], cache consistency techniques have been thoroughly examined in the context of shared memory multiprocessor architectures, which depend on dependable, synchronous broadcast communication facilitated by the system bus. A distributed system, however, may encounter partial failures: a host may malfunction or messages may be lost. Current methodologies for ensuring consistency in file caches can be classified into two categories: those that presume reliable broadcast and hence do not accommodate communication failures, and those that necessitate a consistency verification for each read, thereby resulting in suboptimal performance.

In [10], leases are suggested as a consistency protocol that addresses host and communication failures utilizing physical clocks. An analytical model and evaluation utilizing file access characteristics of the V system demonstrate that short-term leases yield near-optimal efficiency for a broad category of systems, notwithstanding the fault-tolerance provisions. Scholars contend that leases will be more advantageous in future large-scale distributed systems due to their higher ratio of processing speed to network latency and increased aggregate failure rate.

As described by Goel, Miesing, and Chandra [11], the utilization of an extensive dispersed network of machines has emerged as a crucial component of decentralized computing, attributed to the remarkable prevalence of P2P services such as Morpheus, Kazaa, Gnutella, and Napster. Although these systems are primarily recognized for file sharing and associated legal issues, P2P systems are emerging as a highly promising and intriguing field of research. P2P systems provide a distributed, self-sustaining, fault-tolerant, scalable, and symmetric system of machines that effectively balances storage and bandwidth resources. The expansion of the Internet led to the development of distributed file systems. As the host nodes housing the shared items became increasingly geographically dispersed and varied, new criteria and performance limitations such as availability, fault tolerance, security, resilience, and localization techniques emerged as critical considerations in the design of shared file systems.

Benchmarking DCS

Salhi et al. [12] investigated the performance decline of Hazelcast and recommended reducing the number of threads handling client requests to enhance performance. This study sought to conduct a controlled investigation and enhance Yardstick's functionality for performance comparison in multi-client links. Some research has introduced utilities to simplify performance analysis, such as emulators like InterSense, which assist in analyzing the efficiency of decentralized big data applications and enhance sensitivity analysis of intricate decentralized systems. Additionally, Khan et al. [13] have executed experiential performance analyses on decentralized systems, including SQL engines. Numerous studies have suggested various methodologies for executing performance analysis in broad distributed systems.

Tzenetopoulos et al. [14] evaluated the performance of three in-memory information management models: RDD (resilient distributed datasets), Redis, and Mamcached as employed by Spark. The authors conducted a comprehensive performance investigation of object activities, including get and set. Their findings indicate that none of the models effectively manage both workload types. The I/O and CPU efficiency of the TCP stack constituted the bottlenecks for Redis and Memcached. Conversely, owing to the substantial initial costs associated with the get operation, RDD does not facilitate fast retrieval of random objects. Conversely, companies in the decentralized cache models sector, whether commercial or open-source, typically develop metrics to demonstrate their system's performance or to compare it with others, publishing their findings on their websites or as white papers, which may exhibit significant bias.

For instance, Hazelcast [15] developed a benchmark and integrated operations for contrasting their federated cache (3.6-SNAPSHOT) with Red Hat Infinispan 7.2 (a version certified by Red Hat) utilizing the Velocity-detection benchmarking tool. Kathiravelu [16] assert that they are approximately 70% more efficient than Infinispan. This comparison analysis, however, did not consider the volume of simultaneous users, each establishing its own links to the clusters, nor did it include additional recovery processes, such as SQL-based queries.

All the aforementioned tests, with the exception of the one conducted by Agarwal, Hennessy, and Horowitz [17], demonstrate the efficiency factors of cache functions over time, maintaining a constant number of users, a predetermined number of threads per user, and a consistent data volume. While these may suffice in certain instances, they may not

accurately represent actual performance action, where the number of simultaneous users fluctuates automatically throughout the model's runtime. To resolve this problem, Chen, Raab, and Katz [18] devised a technique to accommodate a fluctuating number of simultaneous users and data volumes, with the temporal function, managed by Yarkstick.

Redis in DCS

As described by Sanka, Chowdhury, and Cheung [19], Redis is utilized as the primary database in the cache cluster. Redis is a free and open-source project (licensed under the BSD) that commenced in 2009. It is a C-based, single-threaded NoSQL database. To achieve high performance, Redis employs an in-memory storing method.

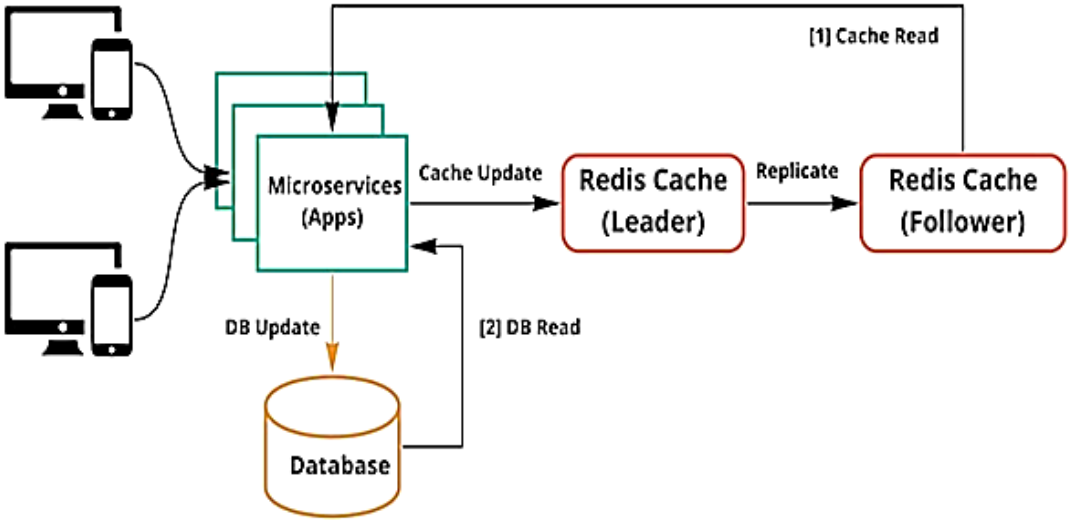

Fig 6. Redis-Based Distributed Caching

Redis is a key-value database characterized by a very simple data structure, which contributes to its superior speed in data operations. Redis accommodates data structures including texts, hashes, lists, sets, and sorted sets. Redis possesses several prominent features, including built-in replication, Lua scripting, and LRU eviction, which enhance its appeal. Redis offers significant reliability and scalability, which are crucial factors in its acceptance. **Fig. 6** illustrates the mechanism of action utilizing the Redis platform. This graphic illustrates the utilization of distributed caching via updates and replication to guarantee enhanced functionality consistently.

IV. EXPERIMENTAL SETUP

System Configuration and Environment

Our testbed was created to measure the overall performance of the proposed linearizable cache versus Redis Cluster with respect to request processing times and fault tolerance. Both regimes were run in Docker containers in an attempt to emulate distributed deployment. All nodes of the system were isolated in their own Docker containers, and this made it possible to control failure conditions and the system behavior during these events rather precisely. The nodes were constrained in a virtualized network and a TCP-proxy was utilized to induce delays and packet loss to the network.

Regarding configuration, they both were run with 3, 5 and 7 nodes, on the same machine, with the help of Docker. The idea behind this setup was to test the scalability as well as the fault tolerance of each of the system whilst doing it under experimental control. Docker was also used to replicate various failure conditions, e.g., disk failures, network delays and network partitions so that a realistic testing environment could be set up in which node failures could be induced and studied.

Benchmarking Methodology and Metrics

To evaluate the performance, two fundamental operations will be benchmarked, the read and write operations. In the read option random keys were sought and the system was asked to retrieve the respective values. This outlined efficiency of important retrieval processes, and the cache eviction policy. Write operations were also tested in which random key-value pairs were inserted or updated to test the capability of the system to synchronize data modifications.

Monitoring latency was done at various percentiles to capture the system performance under different circumstances, i.e., P25, P50, P80, P90, and, P99. Such percentiles indicate a picture of average and worst-case operations at the same time, so an extensive comparison of the systems can be made. This was followed by comparing the fault tolerance capabilities of each of the systems by simulating various possible scenarios of failure and monitoring how each of the systems coped with the consistency and availability issues in such cases.

As shown in **Table 1**, failure simulations were implemented differently where Docker operations simulated disk failures, network delays, and partitions, node failures. Such operations were used in the benchmarking tests to observe the reaction of systems with different failure scenarios.

Table 1. Failure simulation methods

Failure type	Docker operation
Disk failure	Modifying write permissions within a container
Network delay	TCP-proxy command for introducing packet latency
Network partition	Establishing a secondary Docker network and reallocating certain containers to it
Node failure	Terminating the Docker container

Failure Simulation and Scalability

The critical component of experimental arrangement was the failure simulation. Different failure conditions have been simulated by Docker command such as disk failures, network delay, network partitions, and node failure. They were modeled to simulate the actual faults and interpret the capacity of the system to sustain consistency and availability in the face of fault circumstances. We especially paid attention to the behavior of the systems when most of the nodes fail or when the network becomes partitioned.

Scalability was also tested by changing number of nodes in a cluster. To test the scalability characteristics of performance and latency we ran tests with 3, 5 and 7 nodes. **Table 2** and **Table 3** indicate the latencies of read and write operations of the two systems with various configurations.

Table 2. Percentiles of reading request processing time (ms)

Benchmark	P25 (ms)	P50 (ms)	P80 (ms)	P90 (ms)	P99 (ms)
Testing solution	18	30	65	95	171
Redis Cluster	17	26	53	70	150

Table 3. Percentiles of time for processing write requests (ms)

Benchmark	P25	P50	P80	P90	P99
Testing solution	30	49	82	113	230
Redis Cluster	20	31	63	83	162

V. RESULTS AND DISCUSSION

To assess the performance of the planned cache, we contrasted our sequentially consistent cache with Redis Cluster. A well-liked decentralized caching architecture with eventual consistency is Redis Cluster. By definition, a system with eventual consistency can handle requests more quickly than one that is linearizable. We evaluated both systems' fault-tolerance behavior and performance. We created the testing tool using Docker to simulate malfunctioning node activity. Each node is set up using a TCP-proxy in a different docker-container. This enables us to directly create a flawed environment within the test code. The node failure scenarios and how they were implemented are compiled in **Table 1**. The read operation metric for every model is compiled in **Table 2.** The evaluation metric delivers read feedback with a fixed-size value and a randomly generated key. The key eviction element is loaded by introducing random key creation.

Redis Cluster's read time is roughly 20 ms delayed than the suggested option, as can be observed. The uniqueness of the operation application is the cause of this discrepancy. More precisely, a consensus method is invoked for each operation. To ensure that the requested value is pertinent, the invocation is required. Specifically, it is required to retrieve the key value from other devices and retrieved the value that corresponds to the majority.

The internal state of the node is not altered throughout the read operations. On the other hand, write operations entail significant internal node state update processes. We measured the time of processing write requests (see **Table 3**). When compared to Redis Cluster, our system's write operation timing is noticeably slower. The idiosyncrasies of the consensus protocol—the requirement to wait for a reply from most replicas and to write data to disk synchronously—are the cause of this timing discrepancy. To demonstrate the variations in consistency guarantees, we examined how systems behaved under various failure scenarios. The fault tolerance test results are displayed in **Table 4**.

Table 4. Tests for fault tolerance

Solution	Malfunction of the predominant nodes	Elevated latency in single-node operation	Dividing the cluster into two separate networks.	Replica malfunction	Leader malfunction
Tested solution	Failure	Normal operation	Standard cluster functionality inside a network with several nodes	Standard functioning	Standard functioning
Raft Cluster	Normal operation	Normal operation	Existence of dual leaders, data consistency resolution	Data loss	Data loss

The Redis Cluster solution appears susceptible to data delay in the event of a node outage. In case data is archived on the malfunctioning devices, it is quite probable that it will be lost. Moreover, partitioning the system into two groups yields two

Redis Cluster heads, where each copies distinct output records to the accessible devices. Upon network restoration, the cluster enters a non-consistent condition where the user's perception of the data is contingent upon the device from which it retrieves information.

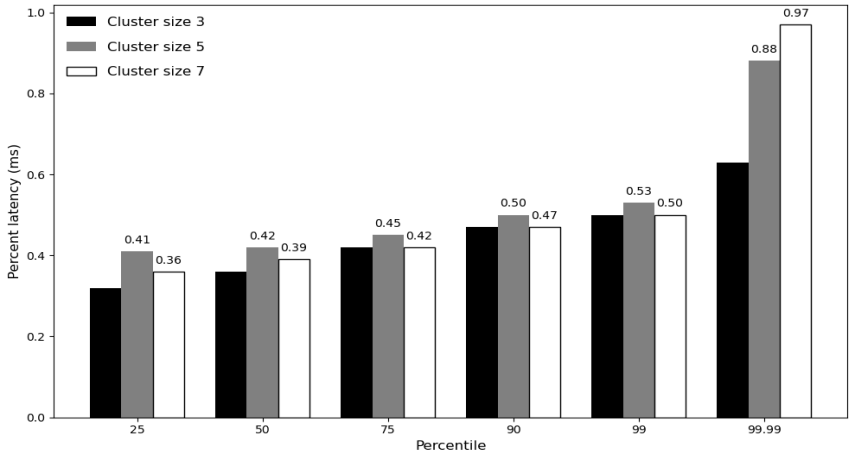

Fig 7. Comparison of Read Operation Percentiles

Our proposed system eliminates these deficiencies by employing a more stringent consistency framework. Thus, our system influences the degree of model accessibility; specifically, if most of the nodes fail, the model under evaluation becomes entirely inaccessible. Specifically, it indicates that the model cannot process feedbacks. Under identical conditions, the Redis Cluster can generate requests in case at least a single node is operational.

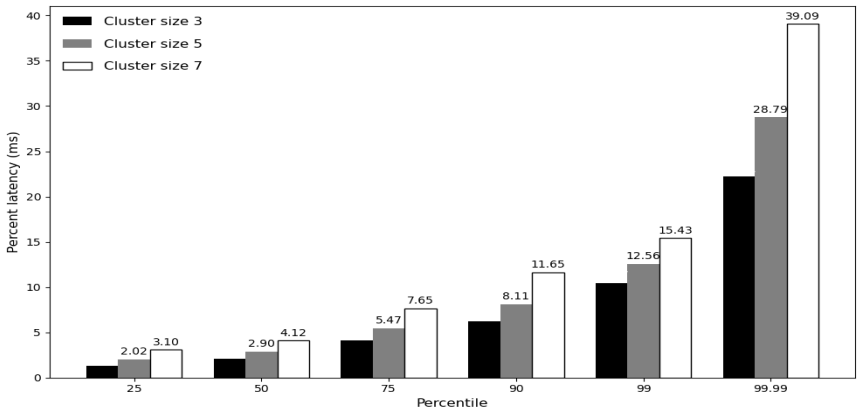

Fig 8. Comparison of Write Operation Distribution Percentiles

According to the evaluation findings, we may delineate the subsequent advantages of our decentralized cache architecture: (i) Robust consistency assurances; and (ii) Reasonable request processing durations, considering the intensive synchronization mechanism. We additionally analyzed the performance of our model across several cluster setups. Read and write activities were evaluated in clusters including 3, 5, and 7 nodes. We evaluated the clusters on an individual tool, with every node operating within a Dockerized environment. **Fig. 7** illustrates the quantiles of the data retrieval process in ms. The read process seems to have significant scalability. The absence or presence of a key in the model does not substantially impact the performance of read operations. To comprehensively assess the influence of cluster volume on performance, we analyzed the distribution points of the write operation. **Fig. 8** illustrates the findings.

The requests durations are significantly longer than those of data retrieval operations. Significantly, our testing container is local, indicating that system delay does not affect the controls. The efficiency of the 7-node group at the 99th percentile is noteworthy. This node group lags behind the other group configurations by over 50 ms. We examined the causes of this elevated delay and determined that the deficiencies are within our storage framework. All 7 nodes were extensively written to the identical base storage system on a singular tool. We acknowledge the limitations of the testing environment for the system. The proposed Docker-driven testing approach is significantly removed from actual cloud settings or production-scale groups. Our testing approach is incapable of simulating all production-level distributed model issues, including optical fiber cable failures and data center outages, among others.

VI. CONCLUSION

We examine the trade-offs among consistency, performance, and fault tolerance in distributed caching systems by comparing a proposed sequentially consistent cache with Redis Cluster. The findings indicate that while the linearizable cache offers enhanced consistency assurances and superior fault tolerance during failures, it results in elevated write latency. Conversely, Redis Cluster offers enhanced availability and expedited read operations, although it is prone to data loss and lacks consistency during failure situations. Scalability test results showed that the two systems were scalable as the cluster size

became larger. The results support the need of proper balance between availability and consistency in distributed systems. Therefore, future research on the system performance should improve the testing conditions and better demonstrate the authenticity of the system in a broad practical setting.

CRediT Author Statement

The author reviewed the results and approved the final version of the manuscript.

Data Availability

The datasets generated during the current study are available from the corresponding author upon reasonable request.

Conflicts of Interests

The authors declare that they have no conflicts of interest regarding the publication of this paper.

Funding

No funding was received for conducting this research.

Competing Interests

The authors declare no competing interests.

References

[1]. R. Alubady, M. Salman, and A. S. Mohamed, "A review of modern caching strategies in named data network: overview, classification, and research directions," Telecommunication Systems, vol. 84, no. 4, pp. 581–626, Sep. 2023, doi: 10.1007/s11235-023-01015-3.

[2]. T. Salah, M. J. Zemerly, N. C. Y. Yeun, M. Al-Qutayri, and Y. Al-Hammadi, "The evolution of distributed systems towards microservices architecture," 2016 11th International Conference for Internet Technology and Secured Transactions (ICITST), pp. 318–325, Dec. 2016, doi: 10.1109/icitst.2016.7856721.

[3]. D. Wilson, "Architecture for a fully decentralized Peer-to-Peer collaborative computing platform," 2015. doi: 10.20381/ruor-4170.

[4]. S. Androutsellis-Theotokis and D. Spinellis, "A survey of peer-to-peer content distribution technologies," ACM Computing Surveys, vol. 36, no. 4, pp. 335–371, Dec. 2004, doi: 10.1145/1041680.1041681.

[5]. Z. Huo et al., "A metadata cooperative caching architecture based on SSD and DRAM for file systems," in Lecture notes in computer science, 2015, pp. 31–51. doi: 10.1007/978-3-319-27122-4_3.

[6]. M. Kowarschik and C. Weiß, "An overview of cache optimization techniques and Cache-Aware numerical algorithms," in Lecture notes in computer science, 2003, pp. 213–232. doi: 10.1007/3-540-36574-5_10.

[7]. Y. Meng, M. A. Naeem, R. Ali, Y. B. Zikria, and S. W. Kim, "DCS: Distributed Caching Strategy at the edge of vehicular sensor Networks in Information-Centric Networking," Sensors, vol. 19, no. 20, p. 4407, Oct. 2019, doi: 10.3390/s19204407.

[8]. S. Borst, V. Gupta, and A. Walid, "Distributed Caching Algorithms for Content Distribution Networks," 2010 Proceedings IEEE INFOCOM, pp. 1–9, Mar. 2010, doi: 10.1109/infcom.2010.5461964.

[9]. F. Petrot, A. Greiner, and P. Gomez, "On cache coherency and memory consistency issues in NOC based shared Memory Multiprocessor SOC architectures," 2022 25th Euromicro Conference on Digital System Design (DSD), pp. 53–60, Jan. 2006, doi: 10.1109/dsd.2006.73.

[10]. Mahmood, R. Exel, H. Trsek, and T. Sauter, "Clock Synchronization over IEEE 802.11—A survey of Methodologies and Protocols," IEEE Transactions on Industrial Informatics, vol. 13, no. 2, pp. 907–922, Dec. 2016, doi: 10.1109/tii.2016.2629669.

[11]. S. Goel, P. Miesing, and U. Chandra, "The impact of illegal Peer-to-Peer file sharing on the media industry," California Management Review, vol. 52, no. 3, pp. 6–33, May 2010, doi: 10.1525/cmr.2010.52.3.6.

[12]. H. Salhi, F. Odeh, R. Nasser, and A. Taweel, "Benchmarking and performance analysis for Distributed cache Systems: A Comparative case study," in Lecture notes in computer science, 2017, pp. 147–163. doi: 10.1007/978-3-319-72401-0_11.

[13]. W. Khan, T. Kumar, C. Zhang, K. Raj, A. M. Roy, and B. Luo, "SQL and NoSQL Database Software Architecture Performance Analysis and Assessments—A Systematic Literature Review," Big Data and Cognitive Computing, vol. 7, no. 2, p. 97, May 2023, doi: 10.3390/bdcc7020097.

[14]. Tzenetopoulos, M. Gazzetti, D. Masouros, C. Pinto, S. Xydis, and D. Soudris, "Disaggregated RDDs: Extending and Analyzing Apache Spark for Memory Disaggregated Infrastructures," 2024 IEEE International Conference on Cloud Engineering (IC2E), pp. 107–117, Sep. 2024, doi: 10.1109/ic2e61754.2024.00019.

[15]. H. Salhi, F. Odeh, R. Nasser, and A. Taweel, "Open Source In-Memory Data Grid Systems: Benchmarking Hazelcast and Infinispan," ICPE '17: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, pp. 163–164, Apr. 2017, doi: 10.1145/3030207.3053671.

[16]. P. Kathiravelu, "An elastic middleware platform for concurrent and distributed cloud and MapReduce simulations," arXiv (Cornell University), Jan. 2016, doi: 10.48550/arxiv.1601.03980.

[17]. Agarwal, J. Hennessy, and M. Horowitz, "Cache performance of operating system and multiprogramming workloads," ACM Transactions on Computer Systems, vol. 6, no. 4, pp. 393–431, Nov. 1988, doi: 10.1145/48012.48037.

[18]. Y. Chen, F. Raab, and R. Katz, "From TPC-C to big Data Benchmarks: a functional workload model," in Lecture notes in computer science, 2013, pp. 28–43. doi: 10.1007/978-3-642-53974-9_4.

[19]. Sanka, M. H. Chowdhury, and R. C. C. Cheung, "Efficient High-Performance FPGA-REDIS Hybrid NoSQL caching System for blockchain scalability," Computer Communications, vol. 169, pp. 81–91, Jan. 2021, doi: 10.1016/j.comcom.2021.01.017.