frontiers | Frontiers in Computer Science

TYPE Original Research
PUBLISHED 30 May 2025
DOI 10.3389/fcomp.2025.1511161

OPEN ACCESS

EDITED BY
Parma Nand,
Auckland University of Technology,
New Zealand

REVIEWED BY
Cheng Yunlong,
Chongqing University of Posts and Telecommunications, China
Yanfeng Jiang,
Jiangnan University, China

*CORRESPONDENCE
Anatoly Sidorov
✉ anatolii.a.sidorov@tusur.ru

RECEIVED 14 October 2024
ACCEPTED 12 May 2025
PUBLISHED 30 May 2025

CITATION
Repin V and Sidorov A (2025) Distributed caching system with strong consistency model.
Front. Comput. Sci. 7:1511161.
doi: 10.3389/fcomp.2025.1511161

Distributed caching system with strong consistency model

Viktor Repin and Anatoly Sidorov*

Department of Data Processing Automation, Tomsk State University of Control Systems and Radioelectronics, Tomsk, Russia

Modern distributed systems store thousands of gigabytes of information in persistent relational database management systems. It is the core storage component for microservice-based architectures. Due to the peculiarities of internal storage mechanisms, the total query processing time may exceed the service level agreement value. A common solution is to add a caching layer between the database management system and the service. However, maintaining the cache in a consistent state across all service replicas is a challenging task. We believe we can design a caching system with a consensus algorithm to meet modern service level agreement requirements. Firstly, we outline the role of caching in a distributed systems context. Secondly, we clarify our consistency model choice. Thirdly, we analyze existing distributed systems and their consistency guarantees to outline the absence of a solution that can fit our requirements. Finally, we develop the architecture of the caching system.

KEYWORDS

caching, distributed systems, linearizability, data consistency, fault tolerance, software architecture, distributed caching, eviction policies

1 Introduction

Caching is widely used in software design and development as an element of optimization under growing data volume. Among other things, it is widely used in fault-tolerant distributed systems to speed up query processing and minimize interaction with persistent DBMSs. Nevertheless, there are many challenges associated with cache implementation in distributed systems. In particular, maintaining strong system consistency with caching, as well as maintaining an acceptable query processing speed, is quite challenging. Existing solutions weaken one of the dimensions. Some works (Hamdard University, 2017; Liu et al., 2019; Pang et al., 2019; Weitzel et al., 2019; Cotroneo et al., 2020; Torabi et al., 2024) utilize subject matter specificity to maintain low query processing latency and strong consistency. Existing general-purpose caching systems such as Redis have quite extensive functionality, but they do not guarantee strong data consistency. For this reason, designing an application software system is quite complex and requires additional financial and human resources to embed caching (Leesatapornwongsa et al., 2016; Ganesan et al., 2017; Berger et al., 2018; Lukman et al., 2019; Strati et al., 2024; Torabi et al., 2024).

The aim of this paper is to close the gap in distributed caching systems with strong consistency. The paper presents the adaptation of Raft's distributed consensus algorithm for our caching system to achieve linearizability. We optimized Raft's customization points with respect to the general features of a general-purpose caching system. Next, we tested our system and its Redis Cluster counterpart in a special environment that can simulate different types of failures. Our results will help to make more informed decisions in the future when implementing caching in distributed fault-tolerant systems. In particular, we believe that our results will help in designing complex microservice-based architectures.

This paper is structured as follows. Section 2 describes the problem of distributed system data consistency and justifies our consistency model choice. Section 3 describes our methodology

to design the architecture of distributed caching. Section 4 describes the results obtained and provides a discussion of the results. Section 5, Conclusion, gives suggestions for practical applications of our results.

2 The role of caching in distributed systems

2.1 Distributed information systems and caching

Modern software products are developed taking into account their possible failure. A failure means that the information system is unavailable for communication for a long period of time. Fault tolerance defines the types and number of failures a system can survive. There are different ways to achieve the desired level of fault tolerance. The basic element of all such mechanisms is replication. Replication allows the system to survive a finite number of failures. It is quite easy to implement in systems that do not store internal state, i.e., data that can affect the processing of a request. This data is stored in a persistent DBMS (Database Management System) that guarantees that all instances of the service operate on the same copy of the data. An example of such a system is applications built on the basis of microservice architecture. Figure 1 shows a model of a distributed application with no internal state.

Figure 1 represents three service instances. Each instance runs the same application business logic. The key point of this architecture is the absence of internal state on service nodes. All state is stored in the persistent DBMS. It should be noted that Figure 1 does not include the network communication protocol as well as the load balancer, since they do not significantly affect the model. This approach has the following advantages:

- The ability to discreetly change the configuration: input and output of service instances;
- High fault tolerance: if N instances of the service are put into operation, the system can survive N-1 failures.

The advantages outlined above make this architectural style the most popular. At the same time, one of its main disadvantages is a high latency when working with a database of large sizes. Cache is a common solution to improve the performance of the previously mentioned model. Cache allows faster interaction with a large-sized database. Figure 2 shows a model that utilizes the caching technique.

The model uses in-memory storage of key-value type as a cache. The model has the advantage that the issue of data consistency is left to the third-party software. Typically, the third-party software is deployed as a separate process. Figure 3 presents a model of such a service.

Such a model adds internal state in the form of a local cache copy to the service. This internal state can lead to data consistency violation. To avoid this, it is necessary to choose a consistency model. The model must satisfy the requirements of distributed storage. As a result, the requirements for the system under development are formulated as follows:

- Transparency to outside observers: a user or a third party sending a request to the system should not be able to distinguish between execution with and without the cache;
- The implemented system must introduce the lowest possible overhead; otherwise, the use of the cache will be unjustified.

Thus, the system should be a distributed data structure, a hash table, with which it will be as easy to work as with the usual, unallocated data structure. Any distributed caching system is designed for strictly defined purposes. For example, the work (Rodriguez et al., 2021) deliberately eschews cache replication in favor of sharding, as the main goal in the work is scaling and load balancing depending on the demand for writes. The main requirement for Google Zanzibar is to preserve the order of user transactions (Liu et al., 2019; Pang et al., 2019). This property is critical for authorization services. As failure to preserve the order can

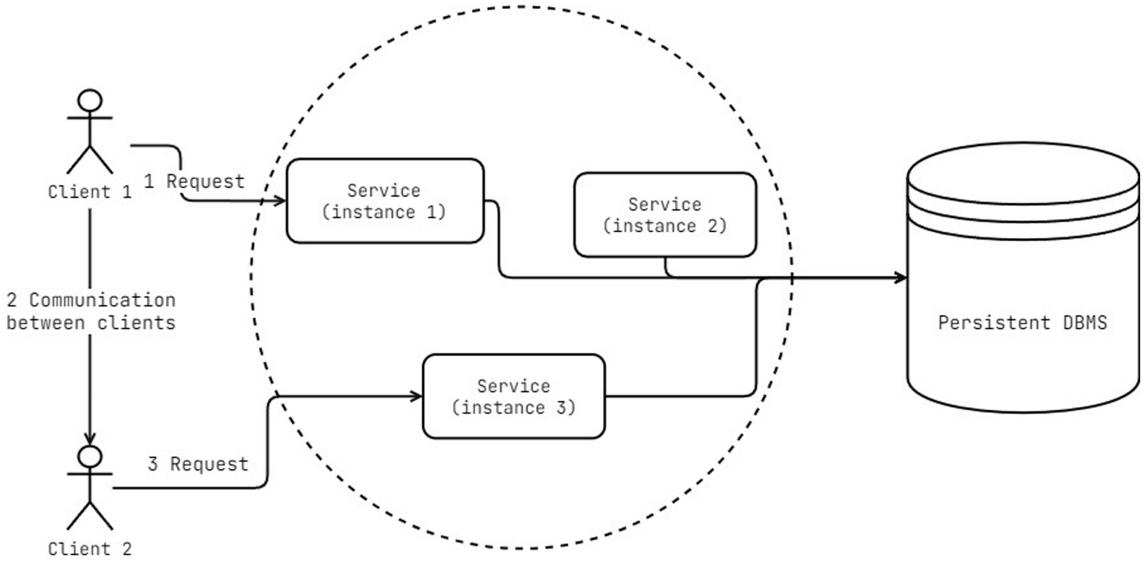

FIGURE 1
A common microservice model.

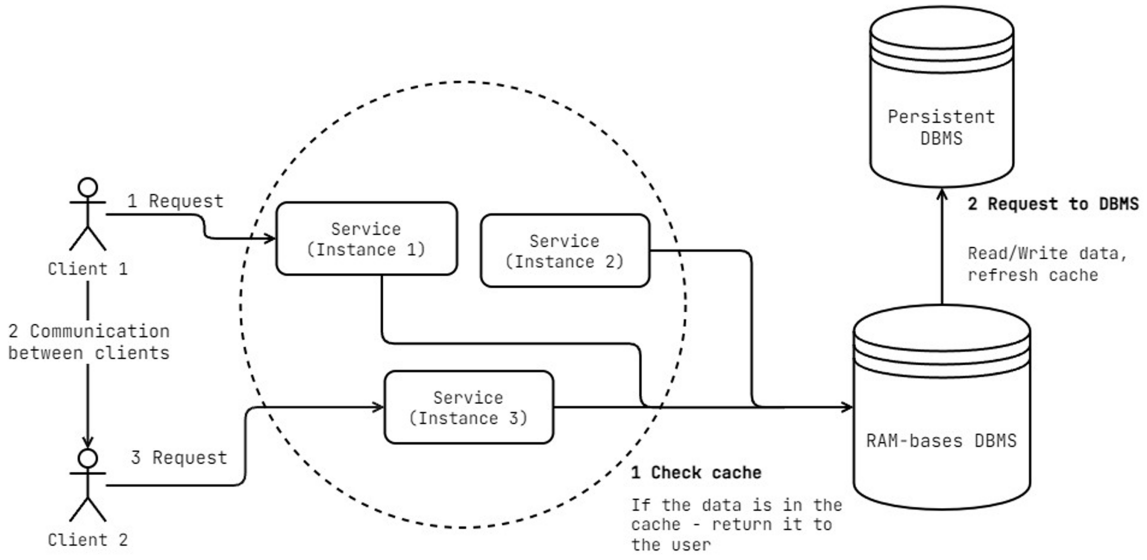

FIGURE 2
The model with the cache.

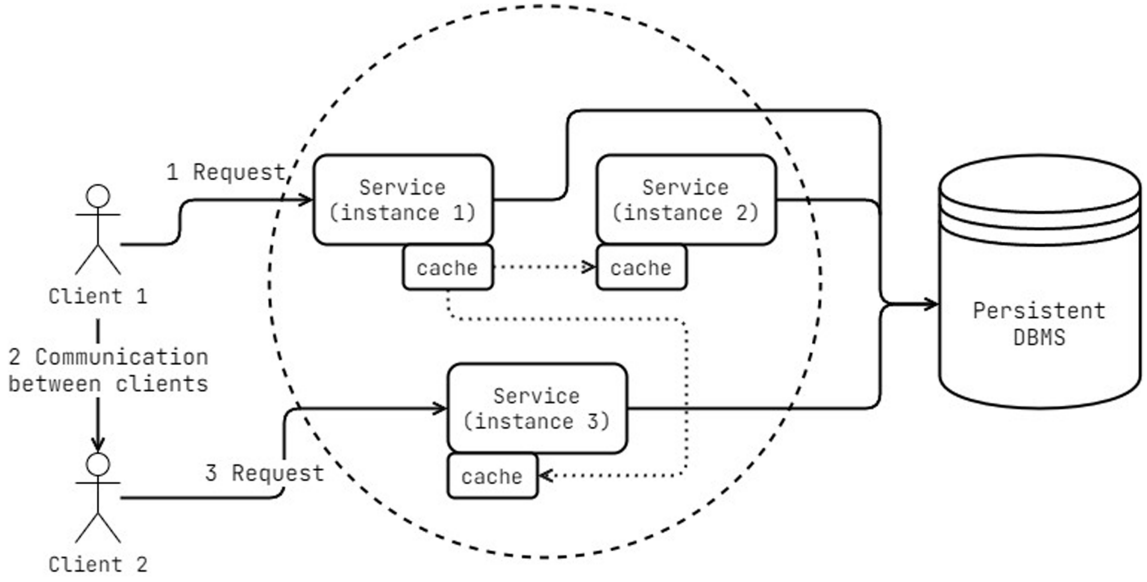

FIGURE 3
The model with the embeded cache.

lead to arbitrary access to sensitive data by an unauthorized user. The Zanzibar system uses caching for "hot" data, where each server replica has a transaction cache. And the entire system provides strict consistency guarantees. However, the cache architecture relies on unique features of the authorization system. For this reason, the Zanzibar cache design cannot be easily adapted to other software solutions.

2.2 Consistency models

The consistency model distinguishes acceptable behavior from incorrect behavior. Behavior is usually described as a history of observed system events. Typical events are reading and writing a value. In addition, the consistency model captures the level of system availability shaped by the following weaknesses of distributed systems:

- Node failure;
- Brain split;
- Node operation delay (node may process a request for a long time, i. e., garbage collector may delay operation or operating system may preempt process from execution).

Explicit identification of such problems is non-trivial and complicates the implementation of the consistency model. At the same

time, the occurrence of any of them in most cases leads to a significant delay in the cluster operation. Therefore, for simplicity in consistency models, a node can be in one of two states: available or unavailable for communication.

Figure 4 shows the relation of consistency models adapted from (Bailis et al., 2013; Viotti and Vukolić, 2017). The left branch of consistency models is of interest for systems implementing distributed transactions. As an example, relational database management systems use distributed transactions. However, distributed transactions are not typical for caches. Therefore, they are not considered here. The models from the right branch are used in various kinds of data warehouses or data processing services. The characteristic of the right branch models is presented in Table 1.

In our case, the main goal is to prefer real-time ordering of operations. It is an essential feature of our system, since the system must modify cache data. In consequence, we chose to use the "Linearizability" model to implement the cache. The main reason behind our decision is the ability of a linearizable system to preserve operation order. Additionally, a linearizable system is composite. This means that we can easily integrate our caching system with other linearizable components. In this way, the whole system will provide linearizable consistency. Note that the practical implementation of any linearizable system is built on a quorum collection operation. To guarantee that a value is reliably stored on replicas, it is necessary to send queries to all cluster members and wait for a response from the majority of nodes. For example, in the case of three servers, it is necessary to wait for the result of two queries; in the case of five, three results are needed, etc. Hence, the system has overhead for network requests at the stage of selecting a consistency model. Therefore, the final system may not be suitable for low-latency software products. An example of such a product is video streaming applications.

2.3 Linearizability

The construction of a linearizable system reduces to solving a consensus problem. The essence of consensus is to choose some value by all nodes of the system. The chosen value must satisfy the properties of agreement, validity, and finiteness/completeness. The agreement means that all nodes have chosen the same value at the next iteration. The finiteness property states that the consensus algorithm must return a result in a deterministic time interval and cannot run indefinitely. Validity is that the value chosen by the algorithm must be offered by some consensus participant.

Currently, there are two main algorithms to solve this problem: Paxos and Raft. The Paxos algorithm is the very first variant of providing consensus. It is used in such DBMSs as Neo4j and Cassandra to select the replication leader. This algorithm has many customization points. For example, Paxos does not require a leader in a cluster, nor does it describe a sequence for changing the composition of its nodes. Paxos distinguishes two node roles: accepting node and offering node. The offering node processes the client request while the receiving node commits the value. The algorithm uses two phases to make a decision: collecting the willing to accept the next value and committing that value. The Paxos algorithm is more of a protocol template. Thus, the final algorithm needs to be thoroughly designed for industry-scale use.

The main advantage of the Raft algorithm is the existence of a clear specification of both the algorithm itself and its implementation. This algorithm fills in the customization points that Paxos has. For example, Raft fixes the stage of replication leader selection. The algorithm defines the following node roles:

- Candidate for becoming a leader in case the current leader fails;
- Replication leader who makes the decisions;

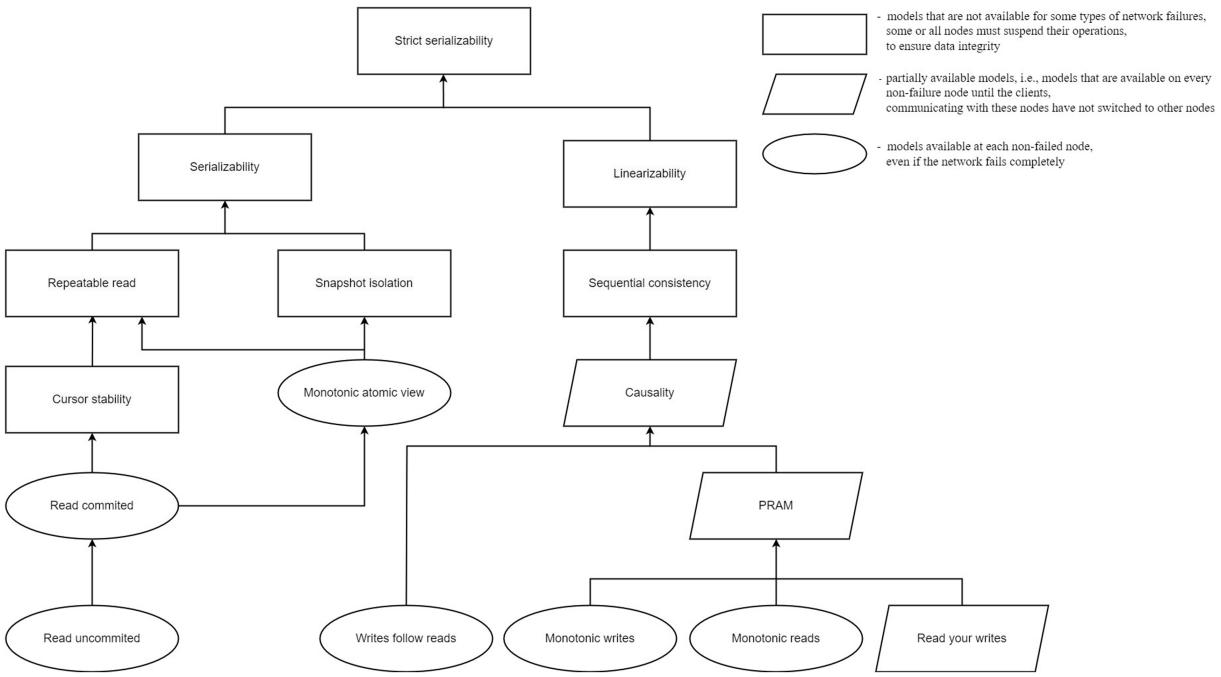

FIGURE 4
The consistency models relationship.

TABLE 1 Description of consistency models.

The model	Description
Read your writes	The model guarantees that a process that makes a write w and then performs a read r will see the write w. This model does not require processes to see each other's records.
Monotonic Writes	The model guarantees that if a process has executed record w1 and then executed record w2, other processes observe record w1 before record w2. This model does not require an order relation between records of different processes.
Monotonic Reads	The model ensures that if a process performs a read r1 and then a read r2, the read r2 cannot observe the records preceding the records reflected in r1, i.e., the read cannot go backwards. Also, like the previous models, monolithic reads do not extend to reads from different processes, focusing only on reads from a single process.
Writes follow reads	The model ensures that if a process reads a value that came about because of a write w1 and then performs a write w2, the effect of w2 must be seen after w1, i.e., once something is read, one cannot change a past read.
Pipeline Random Access Memory	The model is introduced to relax stricter coherent memory models to achieve more efficient and performant parallelism; it ensures that any pair of writes made by one process is observed by any other processes in the order in which they are executed by the executing process, but writes executed by different processes may be observed in different orders.
Causal consistency	The model supports the order of operations. There is a causality important to the application that is observable by all processes, but the order for operations between which there is no causality will not be the same for all processes.
Sequential consistency	The model guarantees that the order of operation execution occurs in some linear sequence. This sequence is consistent with the order of operations on each individual process. However, it does not guarantee linear order among all nodes, i.e., at any given time a process may be overtaking or lagging behind others.
Linearizability	The model guarantees that each operation occurs atomically, in some order consistent with the order of these operations in real time; linearizability is composite: if a system consists of many linearizable modules, then the whole system is linearizable as well.

- Follower, i.e., replication node.

As part of the algorithm's operation, candidates send their identifiers to all nodes, signaling that they are in a working state. In case of leader failure, the candidate sends a special message to the others. This special message signals an attempt to become a cluster leader. Each node responds with an acknowledgment or rejection. The reply message always contains a field with the address of the current leader. Thus, the race of candidates for cluster leader status is eliminated in the system (Ongaro and Ousterhout, 2014). Based on the presented arguments, we can conclude that the Raft algorithm is the most suitable for the realization of the caching system, since there are reference implementations and open-source tools. It allows us to test the correctness of the system based on it.

2.4 Production caching systems

Quite a large number of organizations and enthusiasts are engaged in research and development of distributed caching systems (Hamdard University, 2017; Berger et al., 2018; Abdi et al., 2021; Strati et al., 2024; Torabi et al., 2024). The most well-known industrial variants are Memcached, Redis, Memcache, Dragonfly, and KeyDB. The following criteria were chosen for their analysis:

- Provided consistency model;
- Supported data structures;
- Provided client libraries and SDKs for various programming languages;
- Provided deployment process.

The consistency model allows selecting the appropriate product for the needs of a particular application. Supporting the necessary model at the level of the final product reduces development costs. Hence, when the developer designs or implements the system, it is possible to eliminate the stage of introducing additional components to achieve the necessary consistency guarantees. Supported structures allow increasing the scope of the product application. Because of that, the end user does not have to build the necessary data structures on top of the basic key-value storage. Support for various programming languages allows the systems to be applied to complex software products. It is particularly important since software products may accumulate a large technology stack. The complex deployment model affects the maintenance and support of the software product. When separate deployment units are required to implement a cacheable system, one has to add monitoring of yet another component. This can have a negative impact on the whole system.

Memcached stores key-value data in RAM. It is essentially a server with associative data storage. The product supports client libraries for most programming languages. A server binary and configuration file are required for deployment. Memcached within a single server guarantees the strongest consistency, while maintaining a cluster in the required state falls on the end user. Memcached is the simplest and fastest solution compared to other analogs.

The popular Redis Cluster provides storage with various data structures (strings, lists, hash tables, ordered containers, and geo-reference index). The product uses client–server architecture. For its operation, it is necessary to deploy a separate server to which requests from applications are received. Redis Cluster supports client libraries for most programming languages. The basic version guarantees strong consistency within a single node but does not guarantee it for a cluster of servers. Thus, Redis guarantees finite consistency and cannot be used in linearizable systems. Deployment follows a typical client–server model. There is a separate server consisting of replicas where user data is stored. And there is a client library that allows sending requests to the server.

Memcached, unlike Redis Cluster, has no built-in replication mechanism and is used to implement Memcache-type systems (Cotroneo et al., 2020). The latter works on the principle of third-party cache. The application first checks the presence of data in the cache. In case of a cache miss, it accesses the persistent storage. The main architectural emphasis is on reducing the number of queries to the persistent database. Applications use a special library to interact with the system, hiding all cache logic. Thus, Memcache allows you to independently vary the number of cache servers, replicas, database management system types, and services. This variety allows meeting the needs of a particular application. Memcache supports only the key-value data structure. The main advantage of Memcache over Redis Cluster is the built-in leasing mechanism. Leasing solves a well-known problem in distributed systems called "thundering herd." The problem arises when many threads or processes try to work with a resource. However, only one of these processes can access the resource. In a caching system, it occurs when services detect the absence of the same data and try to insert them into the cache, having received them from an external source beforehand. This situation can offset all the benefits of caching by multiplying the load on the external source. The leasing mechanism eliminates this disadvantage. Memcache facilities issue a special token to one client, which grants the right to only one service to update the cache.

Deployment of the system is non-trivial as it is not a final product unlike Redis and Memcached. Instead, Memcache is a set of utilities and tools on top of Memcached. To deploy it, you need a Memcached executable and a client library for the programming language. You also need to run a program that polls replicas for their state. A database management system-specific plugin is required to reduce the inconsistency window. The plugin sends a delete request to Memcached replicas if data changes.

The KeyDB product is relatively new and is fully compatible with Redis. It differs from its analogs by supporting multithreaded query processing. This feature allows KeyDB to increase throughput. KeyDB supports replication mode with multiple leader nodes, unlike Redis Cluster. It permits writing data from an arbitrary node of the cluster. Backward compatibility with Redis makes it possible to replace KeyDB with Redis Cluster while making minimal changes to the application code. KeyDB does not provide a strict consistency guarantee. However, the project team is trying to introduce linearizability into its software product.

A special feature of the Dragonfly solution is a lock scheduler. The lock scheduler is able to efficiently allocate locks when there is high query competition. This efficiency is achieved through the lightweight concurrency mechanics that Dragonfly uses. It is the main difference with its counterparts that use pessimistic concurrency control (Ren et al., 2015).

All analogs guarantee finite consistency at best. But all of them strive to minimize the window of inconsistency. An example of finite consistency is when a user can read outdated data if he accesses a replica that has not been reached by a deletion request. Redis Cluster tools guarantee consistency if the client is working with the same replica. Otherwise, the consistency model is not documented. Table 2 shows the results of the peer comparison.

An important criterion for choosing any software product is to support target functions "out of the box." This greatly simplifies the development cycle. Within a distributed caching system. It is desirable to support the target consistency model "out of the box," as well as to have a wide set of libraries for application programming and performance analysis. Redis Cluster and KeyDB have such properties. Both systems are complete software products. However, it should be noted that Redis Cluster is more widely used in industrial systems, so its features have been studied and tested in real conditions much more thoroughly. The entire focus of the Memcache architecture is to minimize interaction with the database. For the same reason, Memcache supports only one programming language and has a complex deployment architecture. The KeyDB product is similar to Redis Cluster except for one aspect: KeyDB supports multithreaded query processing. The main disadvantage of KeyDB is the lack of client libraries. All communication is done through a text-based network protocol. The Dragonfly product provides a built-in cluster failure detection mechanism that allows it to automatically recover from failures. Otherwise, Dragonfly is fully compatible with Redis Cluster. None of the counterparts provide strong consistency guarantees, and accordingly, none of them can be used in critical infrastructure systems.

3 The distributed cache architecture

3.1 High-level architecture

Redis Cluster, Memcache, KeyDB, and Dragonfly solutions guarantee eventual consistency of data. The eventual consistency model allows replica data to differ temporarily. Hence, cluster synchronization is achieved only if the system does not receive

TABLE 2 Comparison of caching system analogs.

Solution	Consistency model	Supported data structures	Clients and SDK	Deployment
Memcached	Linearizability among single server	Key-value	The majority of modern programming languages	Single server
Redis Cluster	Eventual consistency	Key-value, list, hash-table.	The majority of modern programming languages	Cluster of servers
Memcache	Eventual consistency	Key-value	PHP	Cluster of servers, service for detecting failures, plugin for DBMS
Dragonfly	Eventual consistency	Key-value, list, hash-table.	Go, Python, Java	Cluster of servers
KeyDB	Linearizability with one leader, eventual consistency at leader failure	Key-value, list, hash-table.	Go, Text-based protocol RESP	Cluster of servers

modification requests for a certain period of time. Otherwise, consistency is guaranteed in case of a client communicating with the same node. Therefore, for a user, the "world picture" is defined by the replica with which he is working. For systems where the causality of actions is important (observance of the order of operations initiated by the user or an external application), the model with eventual consistency is unacceptable. As a result, additional components need to be designed and implemented to achieve the necessary guarantees. Besides, some analogs presented above are not a single product but consist of a set of basic components and additional modules. These components and modules require a full cycle of software development. Based on the comparison of analogs in Table 2, we can formulate the requirements for the mechanisms (tools) of the designed system:

- Support for different eviction policies from the cache;
- Support for different programming languages is required, as the final implementation environment may use different technologies.

The client library architecture of the target system is divided into two levels. This division is due to the fact that the logic of working with the cluster is identical for all programming languages. Only the specificity of the means of expression differs. For example, Python and C# use coroutines and the await operator to work with asynchronous operations, while Java uses threads without special language syntax. The first level of client libraries is implemented in the C++ programming language and contains the basic algorithms and data structures for interacting with replicas of the caching system. The second level is a "wrapper" for the target programming language that works with the first level using cross-lingual interaction mechanisms. Based on the TIOBE rating, three target variants for which "wrappers" are implemented are chosen: Java, C#, and Python. The client and server contain much similar logic. Because of the logic similarity, we designed and implemented a library of common components with the requirements listed below:

- Support for different I/O multiplexing mechanisms; each platform has its own specialized I/O facilities. Not all runtime environments accept third-party networking solutions;
- The ability to change the algorithm for scheduling asynchronous operations and their handlers, taking into account the specifics of the programming language runtime environment, for example, in Python, additional synchronization is required to meet the requirements of the language standard regarding thread safety of the executable code, while in other programming languages such a requirement either does not exist or can be defined by the library user.

Figure 5 shows the class diagram of the I/O module of the library.

The EventLoop class provides a high-level interface to work with the I/O multiplexing mechanism. All system calls must be synchronized, i.e., their competitive execution is prohibited. Violation of this requirement can lead to a program crash at best. Therefore, the EventLoop class hides the synchronization of these system calls behind its interface. A thread is associated with each object of this type. The user must explicitly run one of the Run, RunOnce, or RunForever methods to initiate an event loop. If some I/O logic needs to be scheduled, the RunInEventLoopThread method is provided, whose only parameter is the function scheduled for execution in the event loop thread. The EventLoopNotification-Queue class is used to securely pass handler functions between threads. It is important to note that the RunInEventLoopThread method implements an asynchronous message-passing pattern. The method adds a function to the queue and sends a notification to wake up the event loop if necessary. The notification is sent using the Notifier abstract class. The EventLoopBackend, Event, and Notifier classes hide the platform-dependent implementation of I/O multiplexing mechanisms, thus allowing easy integration of the client library into different programming languages and platforms. For example, the classes are shown in Figure 6.

Depending on the platform, the implementation of some types may vary. For example, for newer versions of the Linux kernel, the io_uring mechanism is supported, which allows efficient operation not only with networking but also with file I/O (Joshi et al., 2024). Therefore, this implementation is preferred but is only available for kernel versions above 5.11. For older kernel versions, the implementation uses system calls of the epoll family to handle networking and a separate thread pool to handle files on the hard disk.

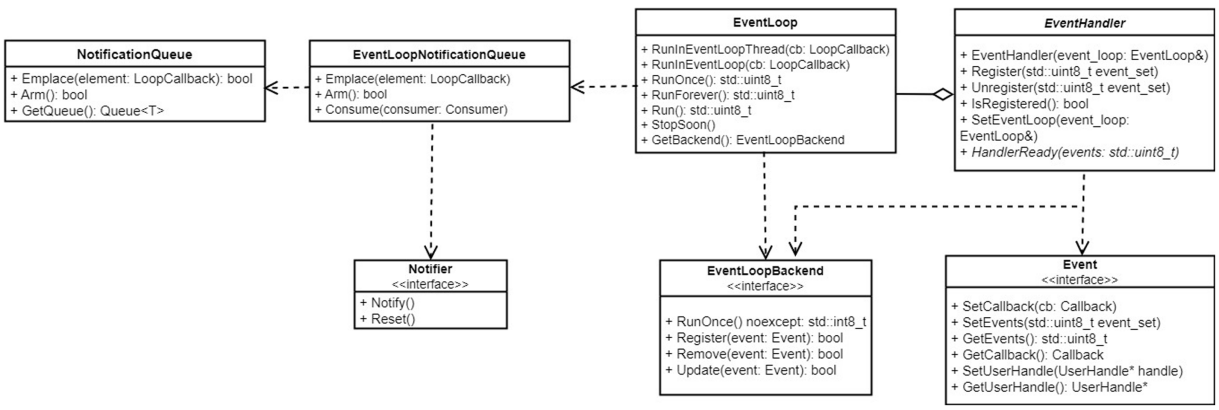

FIGURE 5
The core IO module class diagram.

3.2 The server architecture

The server performs the main data handling operations: replication, eviction of obsolete records, and integrity checking. In addition, the main task of the server is the ability to handle the load generated by users while effectively utilizing the resources of the environment in which it operates. Therefore, the server architecture emphasizes the ability to modify mechanisms for working with external memory, networking, and query execution scheduling algorithms. Figure 7 presents a diagram of the server components.

The client interacts with the server through gRPC technology. The QueryProcessor component contains the logic for processing gRPC calls. To efficiently process incoming requests, QueryProcessor uses the interfaces provided by the core component. The consensus component is responsible for data replication and reliable storage of metainformation for recovery in case of failure. The storage module is used to store data on disk.

Figure 8 shows an interaction diagram that describes the sequence of calls when processing a request from a client.

When a request is received, the following steps are performed. First, the request data is written to disk to allow recovery in case of failure. Second, the data is sent to replicas via the CollectMajority method. Third, a successful response is awaited from the majority. Finally, the response to the client's request is sent. It should be noted that most of the request processing time is spent waiting for a response from the cluster. The consensus module implements the Raft algorithm. The module reliably replicates all cache data, as well as metadata necessary for the eviction algorithm used. The module periodically evicts redundant data from the disk. The consensus module evicts the information about evicted or deleted keys and their values.

To be able to interact with the server, client libraries and tools have been designed in SDK format for different programming languages. There are two main approaches to client SDK development:

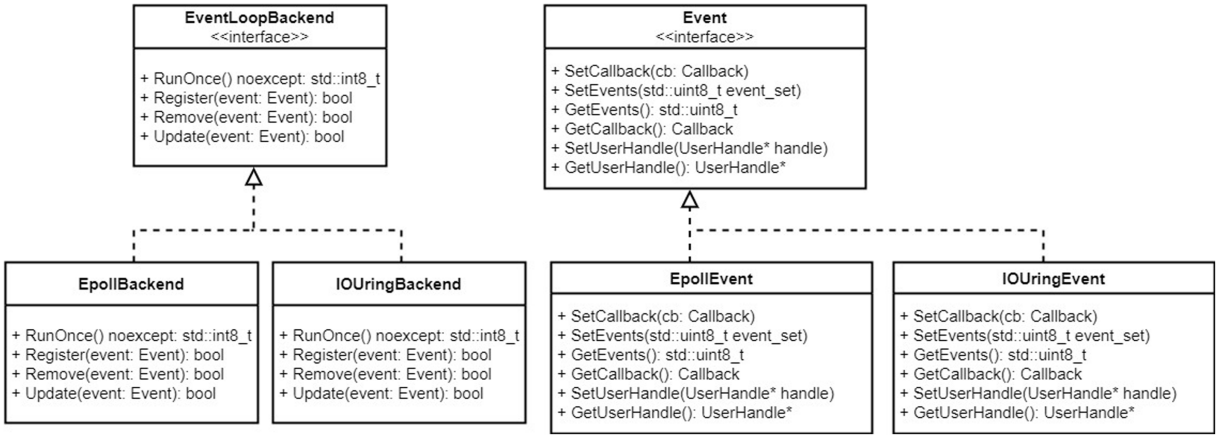

FIGURE 6
Core IO class implemantations.

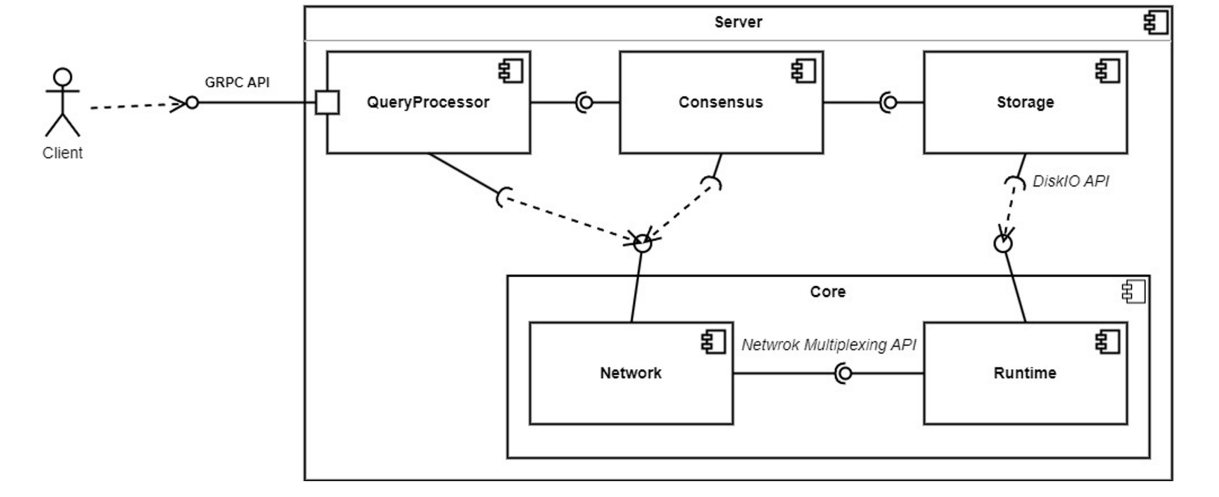

FIGURE 7
Server components.

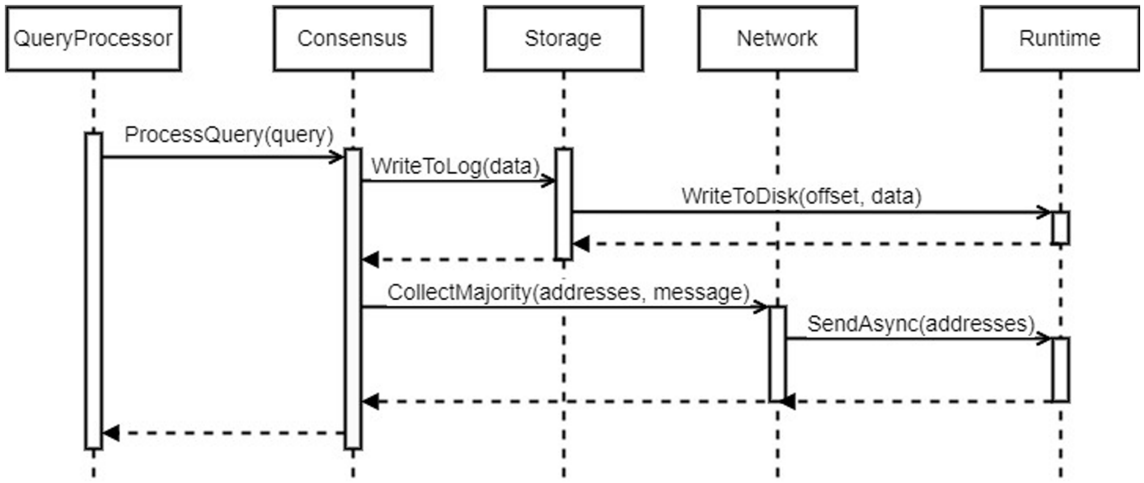

FIGURE 8
Request processing steps.

- Using the network protocol, client libraries are fully implemented in the target programming language;
- Using cross-lingual interaction, the main logic of the client's work is realized in one programming language, and a wrapper is written for other languages using cross-lingual interaction technologies.

We chose the second option. The main reason for this choice is the lower development and release costs of new library versions. The code in the target programming language, in fact, only wraps the main component, so when making changes that do not affect the API, it is enough to replace one file of the dynamic library. Given that all cross-lingual communication mechanisms use dynamic libraries to handle code in a third-party programming language, swapping out an application's dynamic library file is sufficient to release a new version. Figure 9 shows the components of a client library for Python.

As mentioned above, designed wrappers have one goal: to separate the core part implementing communication with a cluster from the part containing language-specific details.

4 Results and discussion

We compared our linearizable cache with Redis Cluster to evaluate the performance of the designed cache. Redis Cluster is a popular distributed caching system with eventual consistency. Eventual consistency, by definition, can process requests faster than a linearizable system. We compared the performance of both systems, and their fault-tolerance behavior. To emulate faulty node operation, we designed the testing tool based on Docker. Every node is deployed in a separate docker-container with a TCP-proxy. This allows us to set up a faulty environment directly in the test code. Table 3 summarizes the node failure cases and their implementation.

Table 4 summarizes the read operation benchmark for each system. The benchmark sends read requests with a randomly generated key and a fixed-size value. The random key generation is introduced to load on the key eviction component.

As can be seen, the read time in the Redis Cluster product is about 20 ms slower than the proposed solution. This difference is due to the peculiarity of the operation implementation. More specifically, every operation involves a consensus algorithm invocation. The invocation is needed to make sure that the requested value is relevant. In particular, it is necessary to find out the value of the key from other nodes and return the value corresponding to the majority.

The read operations do not involve modifications to the node's internal state. Conversely, write operations involve heavy modification steps to the internal node state. We measured write request processing latency (Table 5).

The write operation timing of our system is significantly slower compared to Redis Cluster. This timing difference is due to the peculiarities of the consensus protocol: the need to synchronously write data to disk, as well as to wait for a response from the majority of replicas.

We compared the behavior of systems under different failures to show the difference in consistency guarantees. Table 6 shows the results of the fault tolerance tests.

The Redis Cluster solution seems to be prone to data loss in case of a node failure. More specifically, if data is stored on the failed nodes, it is very likely to be lost. Furthermore, splitting the network into two clusters results in two Redis Cluster leaders, each replicating different records to the available nodes. Once the network is restored, the cluster is in a non-consistent state, and the client's "world picture" depends on the node from which it reads data. The proposed solution is devoid of these shortcomings as it uses a more rigorous consistency model. Consequently, our model affects the level of system availability; i.e., in case of failure of the majority of cluster nodes, the system under test becomes completely unavailable. Particularly, it means the system is unable to process client requests. Under the same conditions, the Redis Cluster is able to process requests if at least one node is functioning. Based on evaluation results, we can outline the following advantages of our distributed cache design:

- Strong consistency guarantees;
- Acceptable request processing timings, taking into account the heavy synchronization algorithm.

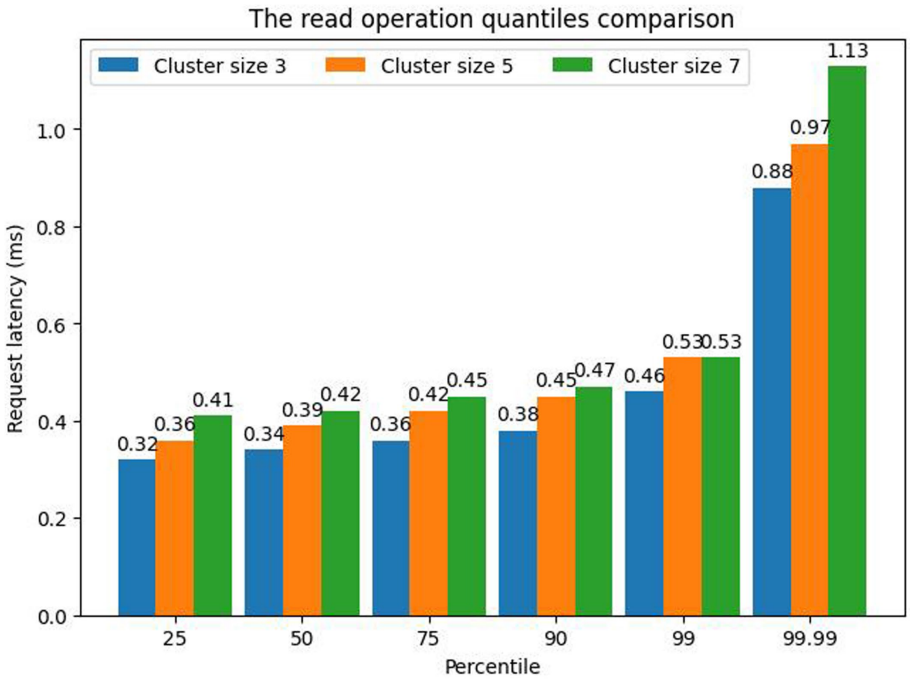

FIGURE 9
The read operation percentiles comparison.

TABLE 3 Failure implementation.

Failure type	Docker operation
Node failure	Stopping the docker container
Network partition	Creating a second docker network and moving some containers to it
Network delay	TCP-proxy command for packet delay
Disk failure	Changing write permissions in a container

TABLE 4 Percentiles of reading request processing time, ms.

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

We also compared the behavior of our system with different cluster configurations. Read and write operations were compared in clusters consisting of three, five, and seven nodes. We tested the cluster on a single machine where each node was running in a docker-container. Figure 9 presents the read operation's quantiles in milliseconds.

The read operation appears to be highly scalable. We noticed that the absence of a key in the system and its presence do not significantly affect the read operation performance. To fully understand the cluster size impact on the performance, we compared the quantiles of the write operation. Figure 10 shows the results.

The request timings are quite higher than those of read operations. Notably, our test environment is local, meaning network latency does not cause the timings. The performance of the seven-node cluster in the ninety-ninth percentile is interesting. This cluster underperforms the other cluster setups by almost 50 milliseconds. Currently, we are investigating the reasons for this high latency. However, we concluded that the main reason flaws in our storage system. All seven nodes were heavily writing to the same underlying storage device on a single machine.

TABLE 5 Percentiles of time for processing write requests, ms.

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

The authors are aware of the shortcomings of the environment in which the system is tested. The presented docker-based testing framework is far from real production-scale clusters or cloud environments. Our testing model cannot simulate all real-world distributed system problems, such as fiber optic cable failure, data center outage, and others. We also recognize the lack of high load in our test environment. We are currently investigating ways to address these shortcomings of our test environment. We have reserved a Kubernetes cluster in Yandex Cloud to deploy our system. The authors are currently developing algorithms and tools to create a load on our system similar to a real production scale environment.

5 Conclusion

The authors of the paper have designed a system whose main purpose is to become an intermediate link between a distributed DBMS and a microservice. This system will improve the query processing time while maintaining strict consistency of data. We designed the runtime part of the cache to make integration more flexible. We have also designed the client application to interact with the caching system. The test results indicated that our system is slightly inferior in performance

TABLE 6 Fault-tolerance tests.

Solution	Leader failure	Replica failure	Splitting the cluster into two networks	High latency in single node operation	Failure of the majority of nodes
Redis Cluster	Data loss	Data loss	Presence of two leaders, data consistency violation	Normal operation	Normal operation
Tested solution	Normal operation	Normal operation	Normal cluster operation in a network with a large number of nodes	Normal operation	Failure

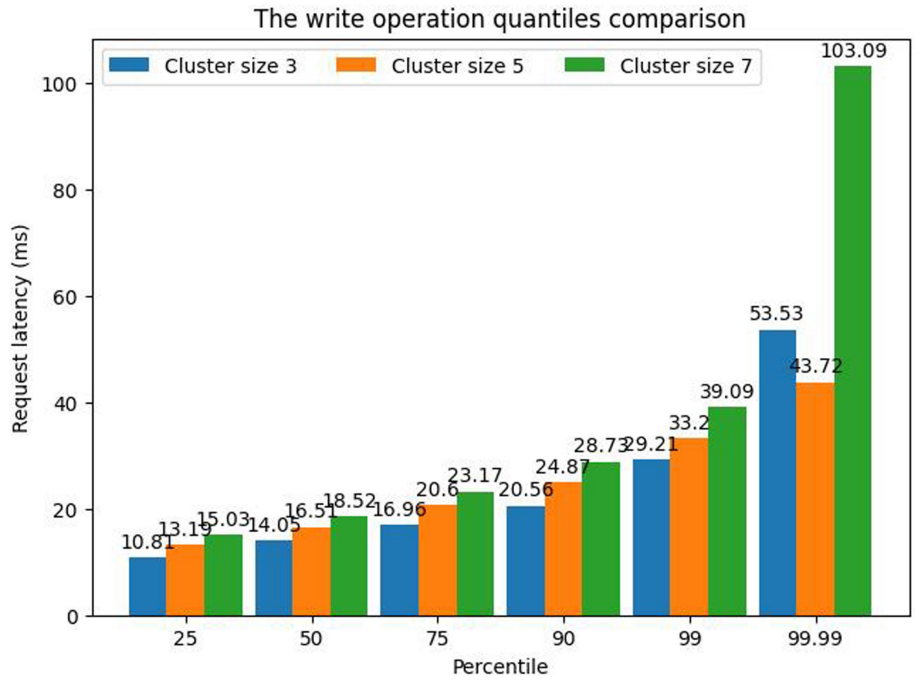

FIGURE 10
The write operation percentiles comparison.

to Redis Cluster. More specifically, if the number of writes is significantly less than the number of reads, our system is almost as efficient as Redis Cluster. These results mean that consensus-based caches can be integrated into production environments with read-heavy loads.

We are planning to investigate more extensively the place of the developed system among analogs and other consistency models. For this purpose, metrics, indicators, and algorithms for their collection are currently being developed. It will help to obtain a more accurate and rigorous picture of the distributed caching system operation.

Data availability statement

The datasets presented in this article are not readily available due to access restrictions. Requests to access the datasets should be directed to hiddenstmail@gmail.com.

Author contributions

VR: Conceptualization, Data curation, Investigation, Resources, Software, Validation, Visualization, Writing – original draft. AS: Conceptualization, Formal analysis, Funding acquisition, Investigation, Methodology, Project administration, Supervision, Validation, Writing – original draft, Writing – review & editing.

Funding

The author(s) declare that financial support was received for the research and/or publication of this article. This research was funded by Ministry of Science and Higher Education of the Russian Federation; project FEWM-2023-0013.

Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Generative AI statement

The author(s) declare that no Gen AI was used in the creation of this manuscript.

Publisher's note

References

Abdi, M., Mosayyebzadeh, A., Hossein, H. M., EUgur, K., Turk, A., Rudolph, L., et al. (2021). "A community cache with complete information." in *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 323–340

Bailis, P., Davidson, A., Fekete, A., Ghodsi, A., Hellerstein, J. M., and Stoica, I. (2013). Highly available transactions: virtues and limitations. *Proc. VLDB Endowment* 7, 181–192. doi: 10.14778/2732232.2732237

Berger, D. S., Berger, D. S., Berg, B., Zhu, T., Harchol-Balter, M., and Sen, S. (2018). "RobinHood: tail latency aware caching – dynamic reallocation from cache-rich to cache-poor" in 13th USENIX symposium on operating systems design and implementation (OSDI 18). ed. C. A. Carlsbad (Carlsbad, CA, USA: USENIX Association), 195–212.

Cotroneo, D., Natella, R., and Rosiello, S. (2020). Dependability evaluation of middleware Technology for Large-scale Distributed Caching. *arXiv*. doi: 10.48550/arXiv.2008.06943

Ganesan, A., Alagappan, R., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. (2017). "Redundancy does not imply fault tolerance: analysis of distributed storage reactions to single errors and corruptions" in 15th USENIX conference on file and storage technologies (FAST 17). ed. C. A. Santa Clara (Santa clara, CA, USA: USENIX Association), 149–166.

Hamdard University (2017). Data cache with distributed cache: a design approach. *Int. J. Comput. Sci. Eng.* 4, 17–23. doi: 10.14445/23488387/IJCSE-V4I6P104

Joshi, K., Gupta, A., Gonz´lez, J., Kumar, A., Kanth Reddy, K., George, A., et al. (2024). "I/O Passthru: upstreaming a flexible and efficient I/O path in Linux." in *22nd USENIX Conference on File and Storage Technologies (FAST 24)*. Santa Clara, CA: USENIX Association, 107–121(Torabi, Khazaei and Litoiu, 2024)

Leesatapornwongsa, T., Lukman, J. F., Lu, S., and Gunawi, H. S. (2016). "TaxDC: a taxonomy of non-deterministic concurrency bugs in datacenter distributed systems." in Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS '16: Architectural support for programming languages and operating systems, Atlanta Georgia USA: ACM, 517–530.

Liu, Z., Bai, Z., Liu, Z., Li, X., Kim, C., Braverman, V., et al. (2019). "DistCache: provable load balancing for large-scale storage systems with distributed caching." in 17th USENIX conference on file and storage technologies (FAST 19). Boston, MA: USENIX Association. pp. 143–157

Lukman, J. F., Suminto, R. O., Simon, D., Priambada, S., Feng Ye, C. T., Leesatapornwongsa, T., et al. (2019). "FlyMC: highly scalable testing of complex Interleavings in distributed systems." in Proceedings of the fourteenth EuroSys conference 2019. EuroSys '19: Fourteenth EuroSys conference 2019, Dresden Germany: ACM, 1–16.

Ongaro, D., and Ousterhout, J. (2014). "In search of an understandable consensus algorithm" in 2014 USENIX annual technical conference (USENIX ATC 14). ed. P. A. Philadelphia (Philadelphia, PA: USENIX Association), 305–319.

Pang, R., Caceres, R., Burrows, M., Chen, Z., Dave, P., Germer, N., et al. (2019). "'Zanzibar: Google's Consistent" in Global authorization system', in 2019 USENIX annual technical conference (Renton, WA: USENIX ATC 19). ed. W. A. Renton (USENIX Association), 33–46.

Ren, K., Thomson, A., and Abadi, D. J. (2015). VLL: a lock manager redesign for main memory database systems. *VLDB J.* 24, 681–705. doi: 10.1007/s00778-014-0377-7

Rodriguez, L. V., Yusuf, F., Lyons, S., Paz, E., Rangaswami, R., Liu, R., et al. (2021) 'Learning cache replacement with CACHEUS', In 19th USENIX conference on file and storage technologies (FAST 21). USENIX Association. pp. 341–354

Strati, F., Mcallister, S., Phanishayee, A., Tarnawski, J., and Klimovic, A. (2024). DéjàVu: KV-cache streaming for Fast, fault-tolerant generative LLM serving'. *arXiv*. doi: 10.48550/arXiv.2403.01876

Torabi, H., Khazaei, H., and Litoiu, M. (2024). "A learning-based caching mechanism for edge content delivery." in *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*. pp. 236–246.

Viotti, P., and Vukolić, M. (2017). Consistency in non-transactional distributed storage systems. *ACM Comput. Surv.* 49, 1–34. doi: 10.1145/2926965

Weitzel, D., Zvada, M., Vukotic, I., Gardner, R., Bockelman, B., Rynge, M., et al. (2019). "StashCache: a distributed caching Federation for the Open Science Grid." in Proceedings of the practice and experience in advanced research computing on rise of the machines (learning), 1–7.