Back To Module Home

Basic Building Blocks for Modern System Design

0% completed

Search Module

Introduction to Building Blocks

Domain Name System

Load balancers

Cache

Databases

Key-value Store

Content Delivery Network (CDN)

Sequencer

Distributed Monitoring

Distributed Cache

- System Design: The Distributed Cache
- Background of Distributed Cache
- High-level Design of a Distributed Cache
- Detailed Design of a Distributed Cache
- Evaluation of a Distributed Cache's Design

Memcached versus Redis

Distributed Messaging Queue

Pub-sub

Rate Limiter

Blob Store

Distributed Search

Distributed Task Scheduler

Sharded Counters

Conclusion

Mark Module as Completed

Memcached versus Redis

Let's compare Memcached and Redis.

We'll cover the following

Introduction#

This lesson will discuss some of the widely adopted real-world implementations of a distributed cache. Our focus will be on two well-known open-source frameworks:

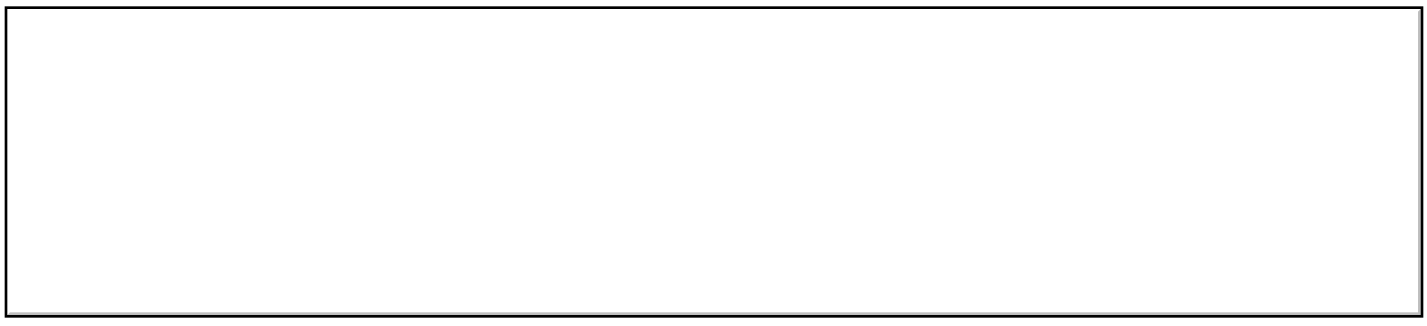
Memcached and Redis. They're highly scalable, highly performant, and robust caching tools. Both of these techniques follow the client-server model and achieve a latency of sub-millisecond. Let's discuss each one of them and then compare their usefulness.

Memcached#

Memcached was introduced in 2003. It's a key-value store distributed cache designed to store objects very fast. Memcached stores data in the form of a key-value pair. Both the key and the value are strings. This means that any data that has been stored will have to be serialized. So, Memcached doesn't support and can't manipulate different data structures.

Memcached has a client and server component, each of which is necessary to run the system. The system is designed in a way that half the logic is encompassed in the server, whereas the other half is in the client. However, each server follows the **shared-nothing architecture**. In this architecture, servers are unaware of each other, and there's no synchronization, data sharing, and communication between the servers.

Due to the disconnected design, Memcached is able to achieve almost a deterministic query speed ($O(1)$) serving millions of keys per second using a high-end system. Therefore, Memcached offers a high throughput and low latency.



Design of a typical Memcached cluster

As evident from the design of a typical Memcached cluster, Memcached scales well horizontally. The client process is usually maintained with the service host that also interacts with the authoritative storage (back-end database).

Facebook and Memcached#

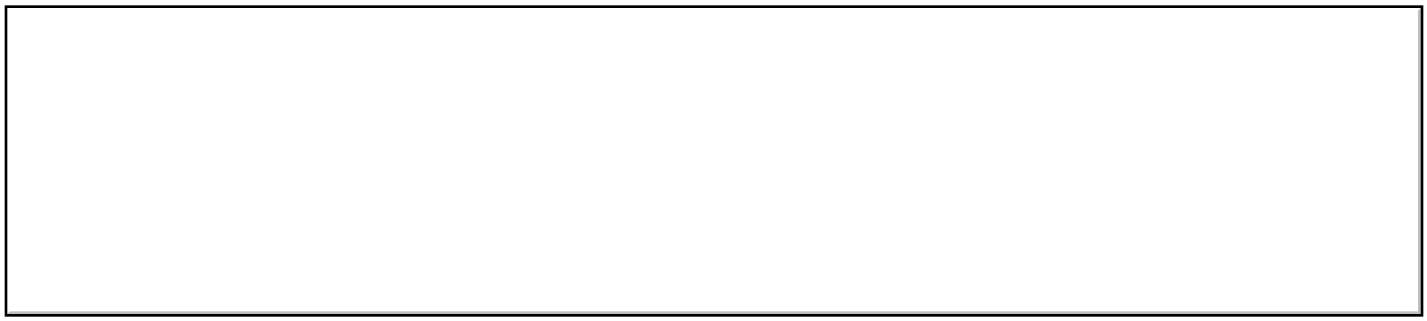
The data access pattern in Facebook requires frequent reads and updates because views are presented to the users on the fly instead of being generated ahead of time. Because Memcached is simple, it was an easy choice for the solution because Memcached started developing in 2003 whereas Facebook was developed in 2004. In fact, in some cases, Facebook and Memcached teams worked together to find solutions.

Some of the simple commands of Memcached include the following:

```
get <key_1> <key_2> <key_3> ...
set <key> <value> ...
delete <key>[<time>] ...
```

At Facebook, Memcached sits between the MySQL database and the web layer that uses roughly 28 TeraBytes of RAM spread across more than 800 servers (as of 2013). By an approximation of least recently used (LRU) eviction policy, Facebook is able to achieve a cache hit rate of 95%.

The following illustration shows the high-level design of caching architecture at Facebook. As we can see, out of a total of 50 million requests made by the web layer, only 2.5 million requests reach the persistence layer.



Facebook using a layer of Memcached sitting between persistence and web layer

Redis#

Redis is a data structure store that can be used as a cache, database, and message broker. It offers rich features at the cost of additional complexity. It has the following features:

- **Data structure store**: Redis understands the different data structures it stores. We don't have to retrieve data structures from it, manipulate them, and then store them back. We can make in-house changes that save both time and effort.
- **Database**: It can persist all the in-memory blobs on the secondary storage.
- **Message broker**: Asynchronous communication is a vital requirement in distributed systems. Redis can translate millions of messages per second from one component to another in a system.

Redis provides a built-in replication mechanism, automatic failover, and different levels of persistence. Apart from that, Redis understands Memcached protocols, and therefore, solutions using Memcached can translate to Redis. A particularly good aspect of Redis is that it separates data access from cluster management. It decouples data and controls the plane. This results in increased reliability and performance. Finally, Redis doesn't provide strong consistency due to the use of asynchronous replication.

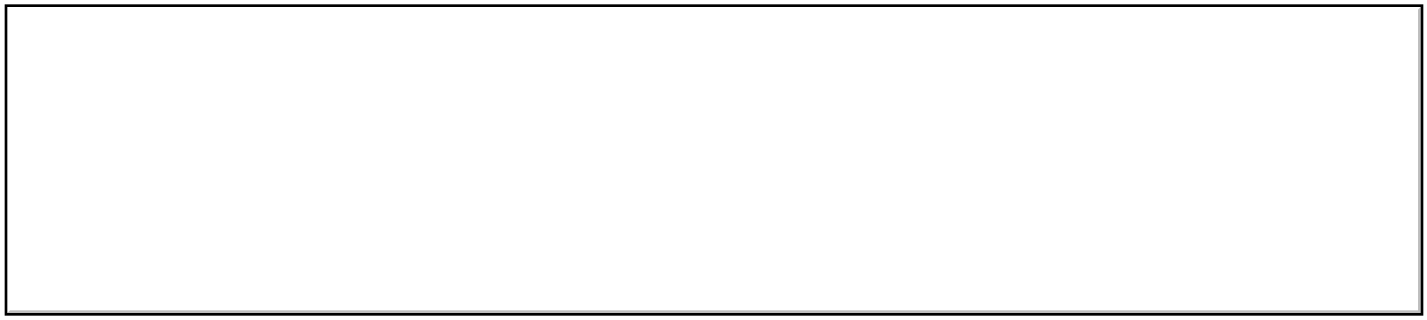
Redis structure supporting automatic failover using redundant secondary replicas

Redis cluster#

Redis has built-in cluster support that provides high availability. This is called Redis

Sentinel. A cluster has one or more Redis databases that are queried using multithreaded proxies. Redis clusters perform automatic sharding where each shard has primary and secondary nodes. However, the number of shards in a database or node is configurable to meet the expectations and requirements of an application.

Each Redis cluster is maintained by a cluster manager whose job is to detect failures and perform automatic failovers. The management layer consists of monitoring and configuration software components.

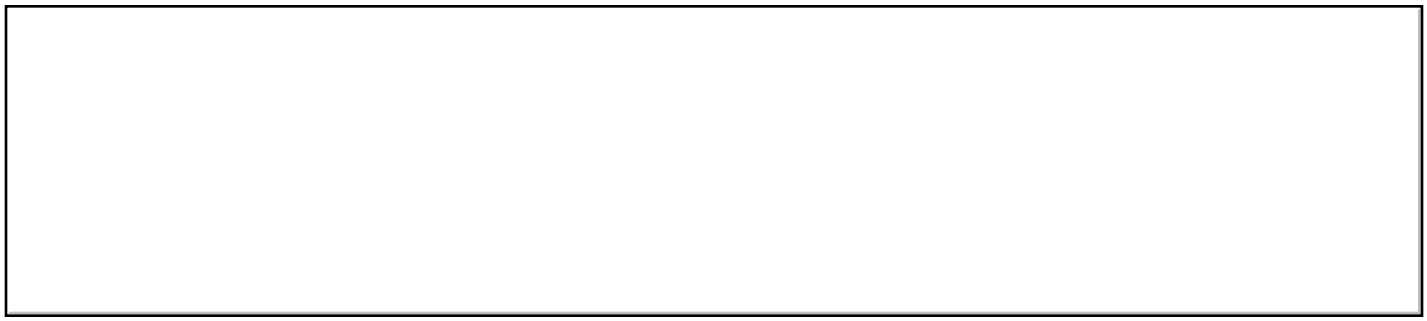


Architecture of Redis clusters

Pipelining in Redis#

Since Redis uses a client-server model, each request blocks the client until the server receives the result. A Redis client looking to send subsequent requests will have to wait for the server to respond to the first request. So, the overall latency will be higher.

Redis uses **pipelining** to speed up the process. **Pipelining** is the process of combining multiple requests from the client side without waiting for a response from the server. As a result, it reduces the number of RTT spans for multiple requests.



Redis client-server communication without pipelining versus Redis client-server communication with

pipelining

The process of pipelining reduces the latency through RTT and the time to do socket level I/O. Also, mode switching through system calls in the operating system is an expensive operation that's reduced significantly via pipelining. Pipelining the commands from the client side has no impact on how the server processes these requests.

For example, two requests pipelined by the client reach the server, and the server can't entertain the second. The server provides a result for the first and returns an error for the second. The client is independent in batching similar commands together to achieve maximum throughput.

> **Note:** Pipelining improves the latency to a minimum of five folds if both the client and server are on the same machine. The request is sent on a loopback address (`127.0.0.1`). The true power of pipelining is highlighted in systems where requests are sent to distant machines.

Memcached versus Redis#

Even though Memcached and Redis both belong to the NoSQL family, there are subtle aspects that set them apart:

- **Simplicity:** Memcached is simple, but it leaves most of the effort for managing clusters left to the developers of the cluster. This, however, means finer control using Memcached. Redis, on the other hand, automates most of the scalability and data division tasks.
- **Persistence:** Redis provides persistence by properties like append only file (AOF) and Redis database (RDB) snapshot. There's no persistence support in Memcached. But this limitation can be catered to by using third-party tools.
- **Data types**: Memcached stores objects, whereas Redis supports strings, sorted sets, hash maps, bitmaps, and hyper logs. However, the maximum key or value size is configurable.

- **Memory usage:** Both tools allow us to set a maximum memory size for caching. Memcached uses the slab allocation method for reducing fragmentation. However, when we update the existing entries' size or store many small objects, there may be a wastage of memory. Nonetheless, there are configuration workarounds to resolve such issues.
- **Multithreading:** Redis runs as a single process using one core, whereas Memcached can efficiently use multicore systems with multithreading technology. We could argue that Redis was designed to be a single-threaded process that reduces the complexity of multithreaded systems. Nonetheless, multiple Redis processes can be executed for concurrency. At the same time, Redis has improved over the years by tweaking its performance. Therefore, Redis can store small data items efficiently. Memcached can be the right choice for file sizes above 100 K.
- **Replication:** As stated before, Redis automates the replication process via few commands, whereas replication in Memcached is again subject to the usage of third-party tools. Architecturally, Memcached can scale well horizontally due to its simplicity. Redis provides scalability through clustering that's considerably complex.

The table below summarizes some of the main differences and common features between Memcached and Redis:

Features Offered by Memcached and Redis

Feature	Memcached	Redis
Low latency	Yes	Yes
Persistence	Possible via third-party tools	Multiple options
Multilanguage support	Yes	Yes
Data sharding	Possible via third-party tools	Built-in solution
Ease of use	Yes	Yes
Multithreading support	Yes	No

Support for data structure	Objects	Multiple data structures
Support for transaction	No	Yes
Eviction policy	LRU	Multiple algorithms
Lua scripting support	No	Yes
Geospatial support	No	Yes

To summarize, Memcached is preferred for smaller, simpler read-heavy systems, whereas Redis is useful for systems that are complex and are both read- and write-heavy.

Points to Ponder

Question 1

Based on the implementation details, which of the two frameworks (Memcached or Redis) has a striking similarity with the distributed cache that we designed in the previous lesson?

Show Answer

Conclusion#

It’s impossible to imagine high-speed large-scale solutions without the use of a caching system. In this chapter, we covered the need for a caching system and its fundamental details, and we orchestrated a basic distributed cache system. We also familiarized ourselves with the design and features of two of the most well-known caching frameworks.

Evaluation of a Distributed Cache's D…

Next

System Design: The Distributed Mess…

Mark as Completed

Report an Issue