Back To Module Home

Basic Building Blocks for Modern System Design

0% completed

Introduction to Building Blocks

Domain Name System

Load balancers

Cache

Databases

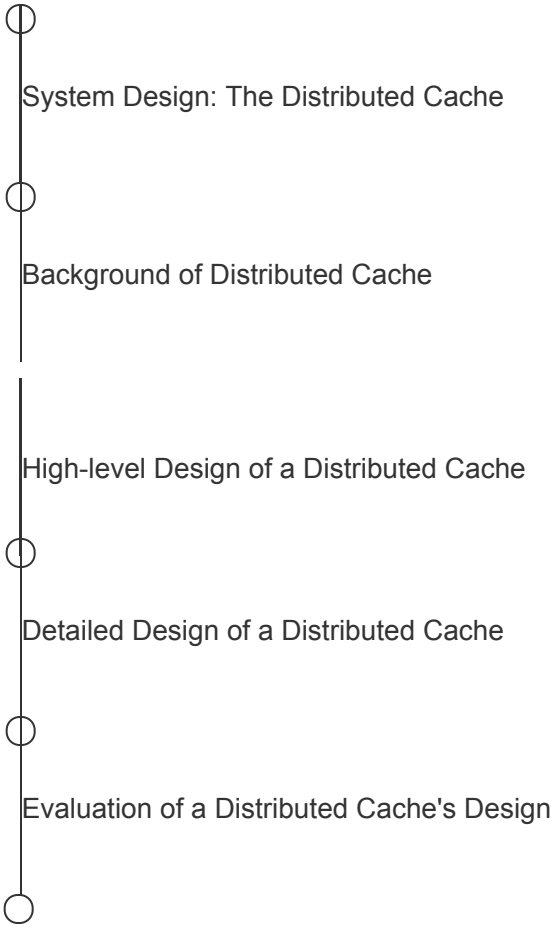
Key-value Store

Content Delivery Network (CDN)

Sequencer

Distributed Monitoring

Distributed Cache



Distributed Messaging Queue

Pub-sub

Rate Limiter

Blob Store

Distributed Search

Distributed Task Scheduler

Sharded Counters

Conclusion

Mark Module as Completed

High-level Design of a Distributed Cache

Learn how we can develop a high-level design of a distributed cache.

We'll cover the following

In this lesson, we'll learn to design a distributed cache. We'll also discuss the trade-offs and design choices that can occur while we progress in our journey towards developing a solution.

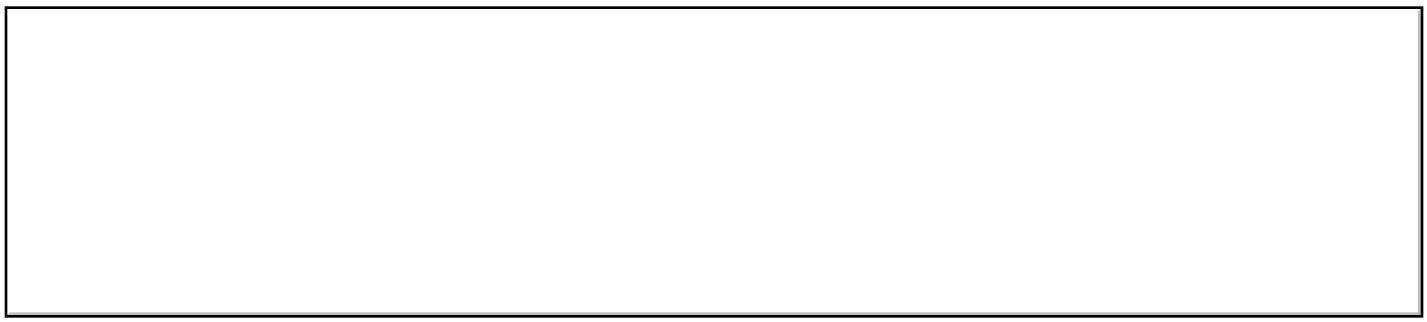
Requirements#

Let us start by understanding the requirements of our solution.

Functional#

The following are the functional requirements:

- **Insert data:** The user of a distributed cache system must be able to insert an entry to the cache.
- **Retrieve data:** The user should be able to retrieve data corresponding to a specific key.



Functional and non-functional requirements of a distributed cache

Non-functional requirements#

We'll consider the following non-functional requirements:

- **High performance**: The primary reason for the cache is to enable fast retrieval of data. Therefore, both the `insert` and `retrieve` operations must be fast.
- **Scalability:** The cache system should scale horizontally with no bottlenecks on an

increasing number of requests.

- **High availability**: The unavailability of the cache will put an extra burden on the database servers, which can also go down at peak load intervals. We also require our system to survive occasional failures of components and network, as well as power outages.

- **Consistency:** Data stored on the cache servers should be consistent. For example, different cache clients retrieving the same data from different cache servers (primary or secondary) should be up to date.

- **Affordability:** Ideally, the caching system should be designed from commodity hardware instead of an expensive supporting component within the design of a system.

API design#

The API design for this problem is sufficiently easy since there are only two basic operations.

Insertion#

The API call to perform insertion should look like this:

```
insert(key, value)
```

Parameter	Description
`key`	This is a unique identifier.
`value`	This is the data stored against a unique `key`.

This function returns an acknowledgment or an error depicting the problem at the server

end.

Retrieval#

The API call to retrieve data from the cache should look like this:

```
retrieve(key)
```

Parameter	Description
`key`	This returns the data stored against the `key`.

This call returns an object to the caller.

Point to Ponder

Question

The API design of the distributed cache looks exactly like the key-value store. What are the possible differences between a key-value store and a distributed cache?

Show Answer

Design considerations#

Before designing the distributed cache system, it's important to consider some design choices. Each of these choices will be purely based on our application requirements. However, we can highlight some key differences here:

Storage hardware#

If our data is large, we may require sharding and therefore use shard servers for cache partitions. Should these shard servers be specialized or commodity hardware? Specialized hardware will have good performance and storage capacity, but it will cost more. We can build a large cache from commodity servers. In general, the number of shard servers will depend on the cache's size and access frequency.

Furthermore, we can consider storing our data on the secondary storage of these servers for persistence while we still serve data from RAM. Secondary storage may be used in cases where a reboot happens, and cache rebuilding takes a long time. Persistence, however, may not be a requirement in a cache system if there's a dedicated persistence layer, such as a database.

Data structures#

A vital part of the design has to be the speed of accessing data. Hash tables are data structures that take a constant time on average to store and retrieve data. Furthermore, we need another data structure to enforce an eviction algorithm on the cached data. In particular, linked lists are a good option (as discussed in the previous lesson).

Also, we need to understand what kind of data structures a cache can store. Even though we discussed in the API design section that we'll use strings for simplicity, it's possible to store different data structures or formats, like hash maps, arrays, sets, and so on, within the cache. In the next lesson, we'll see a practical example of such a cache.

Cache client#

It's the client process or library that places the `insert` and `retrieve` calls. The location of the client process is a design issue. For example, it's possible to place the client process within a serving host if the cache is for internal use only. Otherwise, in the case where the caching system is provided as a service for external use, a dedicated cache client can send users' requests to the cache servers.

Writing policy#

The writing strategy over the cache and database has consistency implications. In general, there's no optimal choice, but depending on our application, the preference of writing policy is significantly important.

Eviction policy#

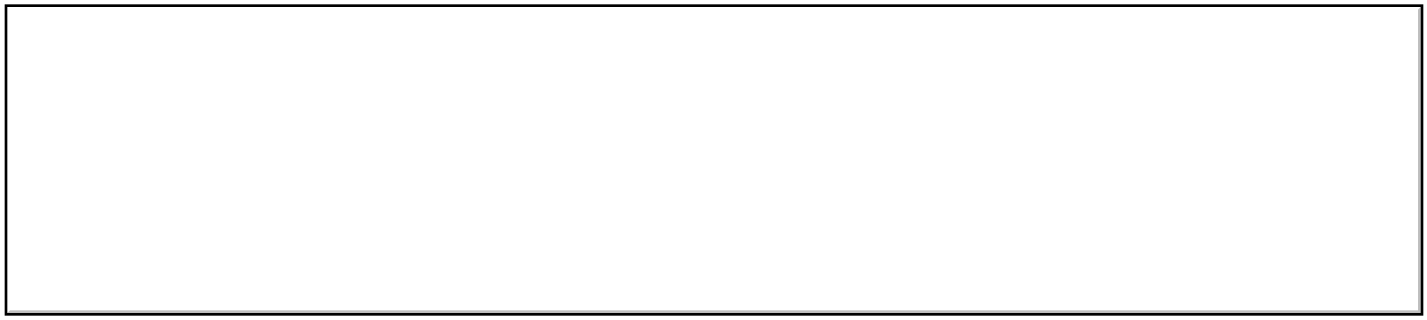
By design, the cache provides low-latency reads and writes. To achieve this, data is often served from RAM memory. Usually, we can't put all the data in the cache due to the limited size of the cache as compared to the full dataset. So, we need to carefully decide what stays in the cache and how to make room for new entries.

With the addition of new data, some of the existing data may have to be evicted from the cache. However, choosing a victim entry depends on the eviction policy. Numerous eviction policies exist, but the choice again depends on the application using it. For instance, least recently used (LRU) can be a good choice for social media services where recently uploaded content will likely get the most views.

Apart from the details in the sections above, optimizing the time-to-live (TTL) value can play an essential role in reducing the number of cache misses.

High-level design#

The following figure depicts our high-level design:



High-level design of distributed cache

The main components in this high-level design are the following:

- **Cache client**: This library resides in the service application servers. It holds all the information regarding cache servers. The cache client will choose one of the cache servers using a hash and search algorithm for each incoming `insert` and `retrieve` request. All the cache clients should have a consistent view of all the cache servers. Also, the resolution technique to move data to and from the cache servers should be the same. Otherwise, different clients will request different servers for the same data.

- **Cache servers**: These servers maintain the cache of the data. Each cache server is accessible by all the cache clients. Each server is connected to the database to store or retrieve data. Cache clients use TCP or UDP protocol to perform data transfer to or from the cache servers. However, if any cache server is down, requests to those servers are resolved as a missed cache by the cache clients.

Background of Distributed Cache

Next

Detailed Design of a Distributed Cache

Mark as Completed

Report an Issue