Consistent Hashing (An Idea to Scale Distributed Systems)

This blog discusses key concepts of consistent hashing, which come in handy while scaling out the distributed storage systems. This idea is frequently applied in designing various real-time applications and solving system design questions during interviews.

What is Consistent Hashing?

Before diving deep into Consistent Hashing, let's first understand what hashing is?

Hashing is a technique used to efficiently retrieve information and improve the performance of insert, delete, and search operations. It involves using a hash function to map data to a storage pool based on key values. For example, we can generate a random number and use it to map data to a server by taking the modulo of the total number of servers. However, this approach is only effective when the number of servers are fixed.

In distributed systems, the number and location of servers may change, making it necessary to have a more efficient way of handling and organizing requests for a scalable application. Consistent hashing is an improvement over normal hashing that addresses these issues.

In consistent hashing, user and server locations are virtually represented in a circular structure called a hash ring. The hash ring is considered infinite and can accommodate any number of servers, which are assigned to random locations based on a hash function. This allows for flexible allocation of servers and efficient handling of requests in a distributed system.

Why do we need Consistent Hashing?

The traditional hashing method is not well-suited for use in distributed

systems where multiple users are requesting services from multiple servers. This is because it assumes that the number of servers and their mapping locations are fixed, which is not the case in a distributed system. If a server fails or goes offline, the traditional method requires significant computation to remap requests to different servers, which can affect the throughput and latency of the service.

In distributed systems, multiple nodes interact with each other and the number of nodes can change due to changes in traffic or maintenance. For example, if we have five nodes in the system and there is an increase in traffic, we may need to add two more nodes to the system, bringing the total to seven. Using traditional hashing, we would need to recompute the mapping of requests, as the hash was previously based on five nodes but now we have seven. Similarly, if the number of nodes decreases due to maintenance or failure, we would need to recompute the mapping again. This is inefficient and requires a lot of redundant computation and data reshuffling within the cluster.

In situations where the number of operational servers is not fixed, the traditional hashing method becomes increasingly inefficient as the number of servers increases, since it requires more and more computation and reassignment of requests. To address these shortcomings, the concept of consistent hashing was developed as a more dynamic solution.

How Consistent Hashing Works?

Consistent Hashing helps us in effective organization and distribution of resources by ensuring minimum reorganization of requests in any failure. In Consistent Hashing, a hash function is used to map servers to locations in a virtual ring. The position of the server is just a random position obtained using the hash function.

Consistent Hashing is organized in the following manner:

1. The servers are hashed using their IP addresses and assigned

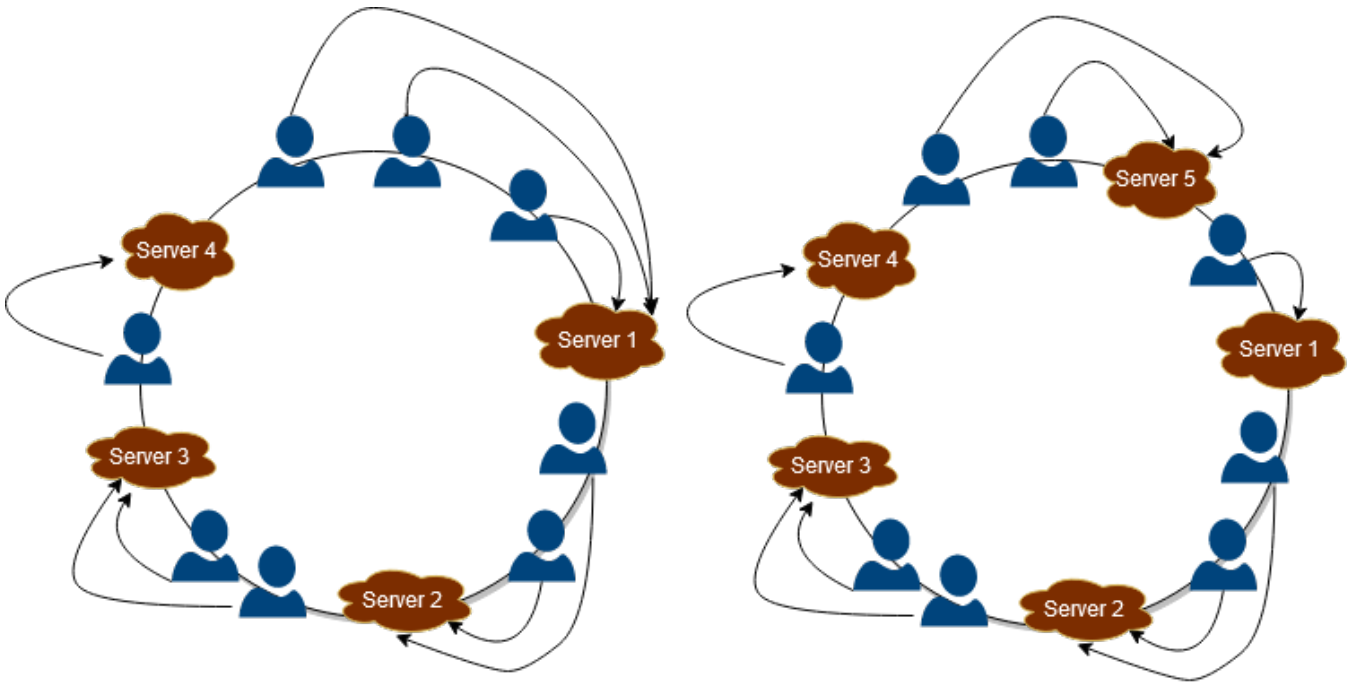
positions based on the output of hash function.

2. Similarly, keys are hashed to positions using the same hash function and placed in the virtual ring.
3. They map keys with the server having the same position, and in case the position doesn't match, then assign the key to the first server that we get while moving in a clockwise direction.

So in this manner, the keys are assigned to the server in Consistent Hashing. The beauty of Consistent Hashing comes when we add or remove servers.

Addition of new server

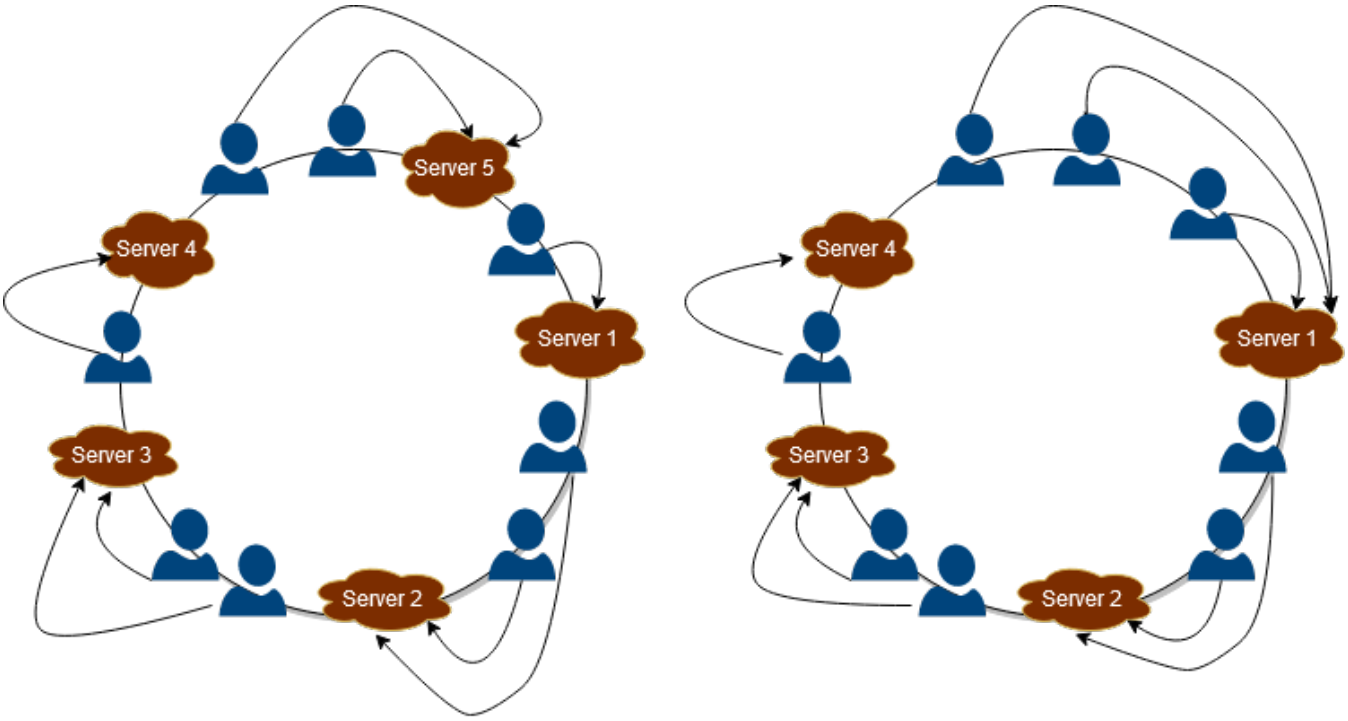
When a new server is added to the application, it is mapped using the hash function and allocated to the hash ring's desired location. After its allotment, all the keys will map on these newly added servers passing its location. This is depicted in the figure below. When server 5 is added between 1and 4, all the requests after 4 are assigned to 5 instead of mapping to 1. Hence in this way, Consistent Hashing helps reduce loads of massive servers and proves highly effective in scaling and increasing the throughput, and improves the latency of the application.



Addition of new server

Removal of server

Whenever any server fails in the system, then all the keys previously mapped to the failed server will redirect to the next server, which is located after the failed server in the clockwise direction. Hence in this manner, the service remains active and provides fault-tolerant service. This is depicted in the figure below. When server 4 breakdowns, then all the keys mapped to 4 are reallocated to 1, preventing the system from breaking down.

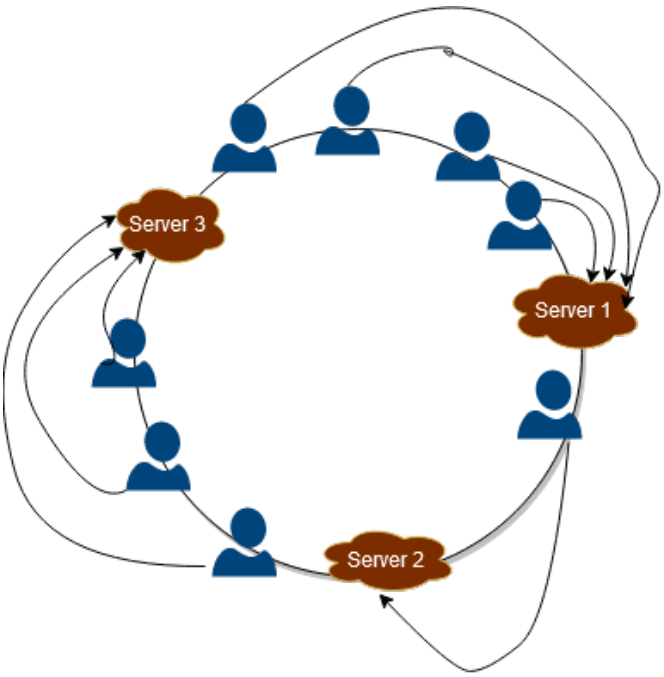


Removal of server

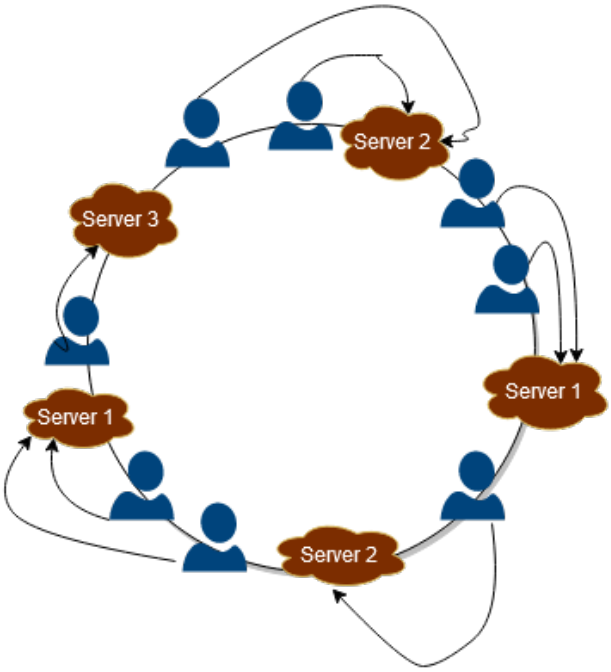
Non-Uniform Distribution

There is a shortcoming of this approach. All the keys may get mapped to the same server, and hence one server will get all the workload, and all the other servers will remain idle. This situation is very inefficient and is very prone to failure. To deal with this, a new concept has been introduced. All the servers are replicated and arranged at different positions in the ring. In this manner, with an increased number of servers, the distribution becomes much more uniform and helps in the service's scaling. This is depicted in the figure below. All the servers are replicated and allocated to different locations, and hence this makes the distribution of keys uniform

in the hash ring.



Non Uniform Distribution



Replication

Conclusion

Consistent Hashing is one of the most crucial concepts in designing distributed systems as it tackles the scalability challenges with dynamic nodes assignment and provides fault tolerance. It is also very useful in system design interviews. This concept allows the distribution of requests or data in the servers and their mapping to servers efficiently. It helps in achieving Horizontal Scaling and increases the throughput and improves the latency of the application.

Thanks to Suyash for his contribution in creating the first version of this content. If you have any queries/doubts/feedback, please write us at contact@enjoyalgorithms.com. Enjoy learning, Enjoy system design, Enjoy algorithms!