• A new request arrives at 1:00:30, the timestamp 1:00:30 is inserted into the log. After the insertion, the log size is 2, not larger than the allowed count. Thus, the request is allowed.

• A new request arrives at 1:00:50, and the timestamp is inserted into the log. After the insertion, the log size is 3, larger than the allowed size 2. Therefore, this request is rejected even though the timestamp remains in the log.

• A new request arrives at 1:01:40. Requests in the range [1:00:40,1:01:40) are within the latest time frame, but requests sent before 1:00:40 are outdated. Two outdated timestamps, 1:00:01 and 1:00:30, are removed from the log. After the remove operation, the log size becomes 2; therefore, the request is accepted.

Pros:

• Rate limiting implemented by this algorithm is very accurate. In any rolling window, requests will not exceed the rate limit.

Cons:

• The algorithm consumes a lot of memory because even if a request is rejected, its timestamp might still be stored in memory.

Sliding window counter algorithm

The sliding window counter algorithm is a hybrid approach that combines the fixed window counter and sliding window log. The algorithm can be implemented by two different approaches. We will explain one implementation in this section and provide reference for the other implementation at the end of the section. Figure 4-11 illustrates how this algorithm works.

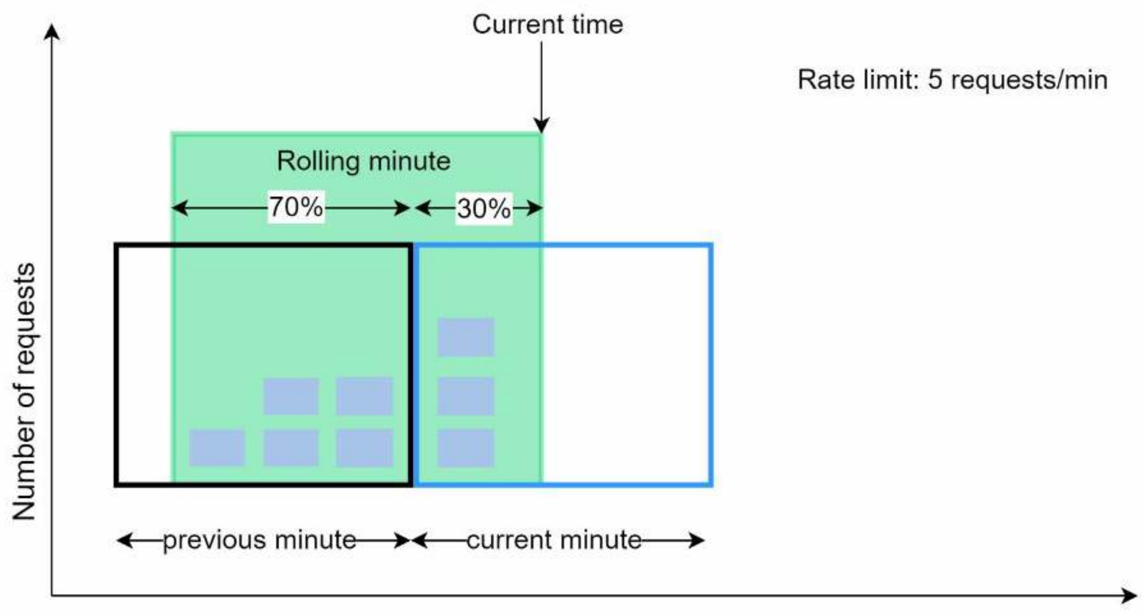


Figure 4-11

Assume the rate limiter allows a maximum of 7 requests per minute, and there are 5 requests in the previous minute and 3 in the current minute. For a new request that arrives at a 30% position in the current minute, the number of requests in the rolling window is calculated using the following formula:

• Requests in current window + requests in the previous window * overlap percentage of the rolling window and previous window

- Using this formula, we get 3 + 5 * 0.7% = 6.5 request. Depending on the use case, the number can either be rounded up or down. In our example, it is rounded down to 6.

Since the rate limiter allows a maximum of 7 requests per minute, the current request can go through. However, the limit will be reached after receiving one more request.

Due to the space limitation, we will not discuss the other implementation here. Interested readers should refer to the reference material [9]. This algorithm is not perfect. It has pros and cons.

Pros

- It smooths out spikes in traffic because the rate is based on the average rate of the previous window.
- Memory efficient.

Cons

- It only works for not-so-strict look back window. It is an approximation of the actual rate because it assumes requests in the previous window are evenly distributed. However, this problem may not be as bad as it seems. According to experiments done by Cloudflare [10], only 0.003% of requests are wrongly allowed or rate limited among 400 million requests.

High-level architecture

The basic idea of rate limiting algorithms is simple. At the high-level, we need a counter to keep track of how many requests are sent from the same user, IP address, etc. If the counter is larger than the limit, the request is disallowed.

Where shall we store counters? Using the database is not a good idea due to slowness of disk access. In-memory cache is chosen because it is fast and supports time-based expiration strategy. For instance, Redis [11] is a popular option to implement rate limiting. It is an in-memory store that offers two commands: INCR and EXPIRE.

- INCR: It increases the stored counter by 1.
- EXPIRE: It sets a timeout for the counter. If the timeout expires, the counter is automatically deleted.

Figure 4-12 shows the high-level architecture for rate limiting, and this works as follows:

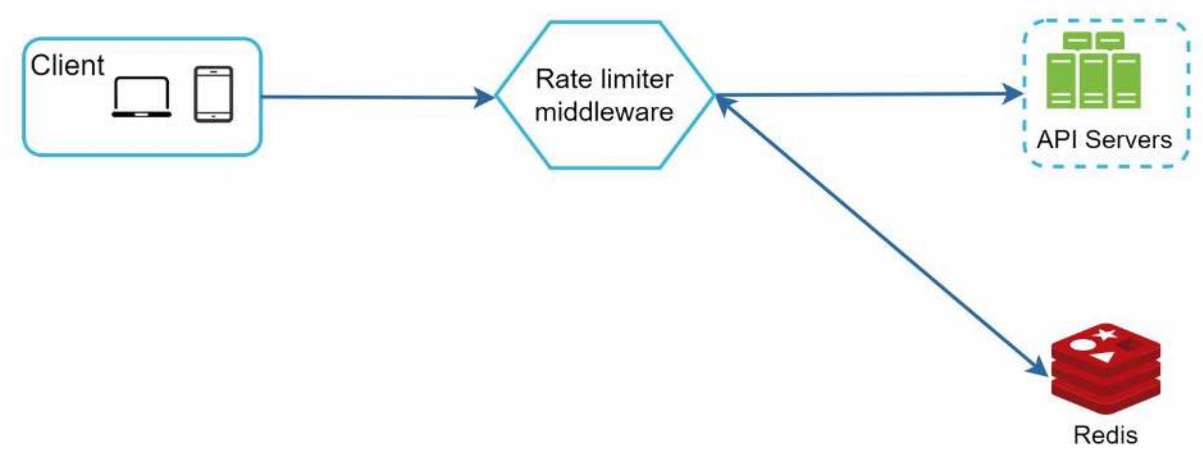


Figure 4-12

- The client sends a request to rate limiting middleware.
- Rate limiting middleware fetches the counter from the corresponding bucket in Redis and

checks if the limit is reached or not.

- If the limit is reached, the request is rejected.
- If the limit is not reached, the request is sent to API servers. Meanwhile, the system increments the counter and saves it back to Redis.

Step 3 - Design deep dive

The high-level design in Figure 4-12 does not answer the following questions:

- How are rate limiting rules created? Where are the rules stored?
- How to handle requests that are rate limited?

In this section, we will first answer the questions regarding rate limiting rules and then go over the strategies to handle rate-limited requests. Finally, we will discuss rate limiting in distributed environment, a detailed design, performance optimization and monitoring.

Rate limiting rules

Lyft open-sourced their rate-limiting component [12]. We will peek inside of the component and look at some examples of rate limiting rules:

```
domain: messaging
descriptors:
  - key: message_type
    Value: marketing
    rate_limit:
      unit: day
      requests_per_unit: 5
```

In the above example, the system is configured to allow a maximum of 5 marketing messages per day. Here is another example:

```
domain: auth
descriptors:
  - key: auth_type
    Value: login
    rate_limit:
      unit: minute
      requests_per_unit: 5
```

This rule shows that clients are not allowed to login more than 5 times in 1 minute. Rules are generally written in configuration files and saved on disk.

Exceeding the rate limit

In case a request is rate limited, APIs return a HTTP response code 429 (too many requests) to the client. Depending on the use cases, we may enqueue the rate-limited requests to be processed later. For example, if some orders are rate limited due to system overload, we may keep those orders to be processed later.

Rate limiter headers

How does a client know whether it is being throttled? And how does a client know the number of allowed remaining requests before being throttled? The answer lies in HTTP response headers. The rate limiter returns the following HTTP headers to clients:

X-Ratelimit-Remaining: The remaining number of allowed requests within the window.

X-Ratelimit-Limit: It indicates how many calls the client can make per time window.

X-Ratelimit-Retry-After: The number of seconds to wait until you can make a request again without being throttled.

When a user has sent too many requests, a 429 too many requests error and *X-Ratelimit-*

Retry-After header are returned to the client.

Detailed design

Figure 4-13 presents a detailed design of the system.

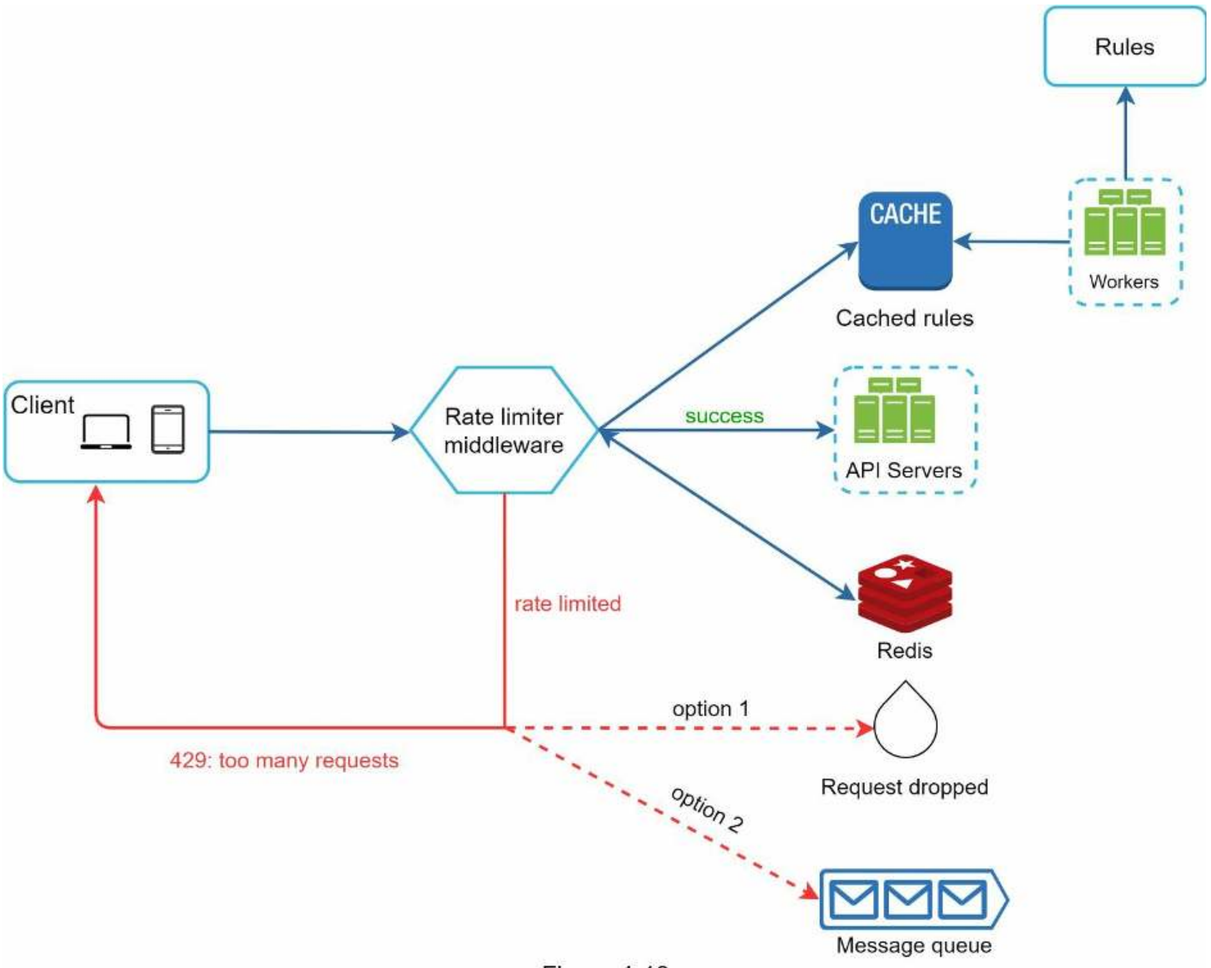


Figure 4-13

- Rules are stored on the disk. Workers frequently pull rules from the disk and store them in the cache.
- When a client sends a request to the server, the request is sent to the rate limiter middleware first.
- Rate limiter middleware loads rules from the cache. It fetches counters and last request timestamp from Redis cache. Based on the response, the rate limiter decides:
    - if the request is not rate limited, it is forwarded to API servers.
    - if the request is rate limited, the rate limiter returns 429 too many requests error to the client. In the meantime, the request is either dropped or forwarded to the queue.

Rate limiter in a distributed environment

Building a rate limiter that works in a single server environment is not difficult. However, scaling the system to support multiple servers and concurrent threads is a different story. There are two challenges:

- Race condition
- Synchronization issue

Race condition

As discussed earlier, rate limiter works as follows at the high-level:

- Read the *counter* value from Redis.
- Check if (*counter + 1*) exceeds the threshold.
- If not, increment the counter value by 1 in Redis.

Race conditions can happen in a highly concurrent environment as shown in Figure 4-14.

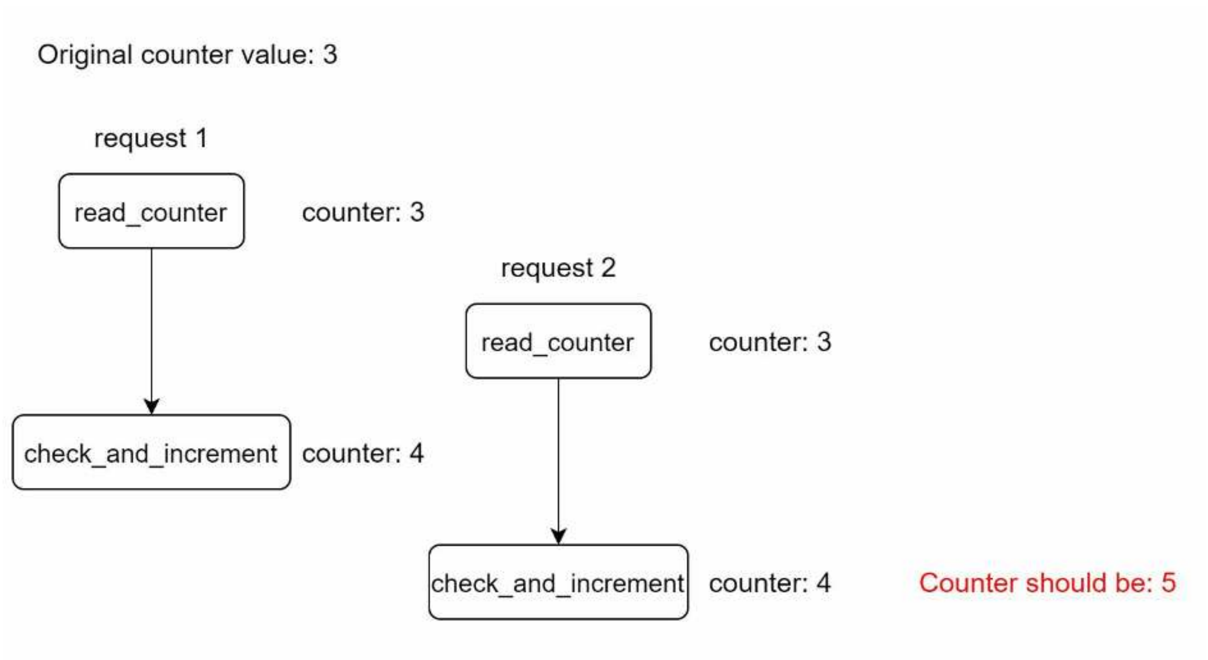


Figure 4-14

Assume the *counter* value in Redis is 3. If two requests concurrently read the *counter* value before either of them writes the value back, each will increment the *counter* by one and write it back without checking the other thread. Both requests (threads) believe they have the correct *counter* value 4. However, the correct *counter* value should be 5.

Locks are the most obvious solution for solving race condition. However, locks will significantly slow down the system. Two strategies are commonly used to solve the problem: Lua script [13] and sorted sets data structure in Redis [8]. For readers interested in these strategies, refer to the corresponding reference materials [8] [13].

Synchronization issue

Synchronization is another important factor to consider in a distributed environment. To support millions of users, one rate limiter server might not be enough to handle the traffic. When multiple rate limiter servers are used, synchronization is required. For example, on the left side of Figure 4-15, client 1 sends requests to rate limiter 1, and client 2 sends requests to rate limiter 2. As the web tier is stateless, clients can send requests to a different rate limiter as shown on the right side of Figure 4-15. If no synchronization happens, rate limiter 1 does not contain any data about client 2. Thus, the rate limiter cannot work properly.

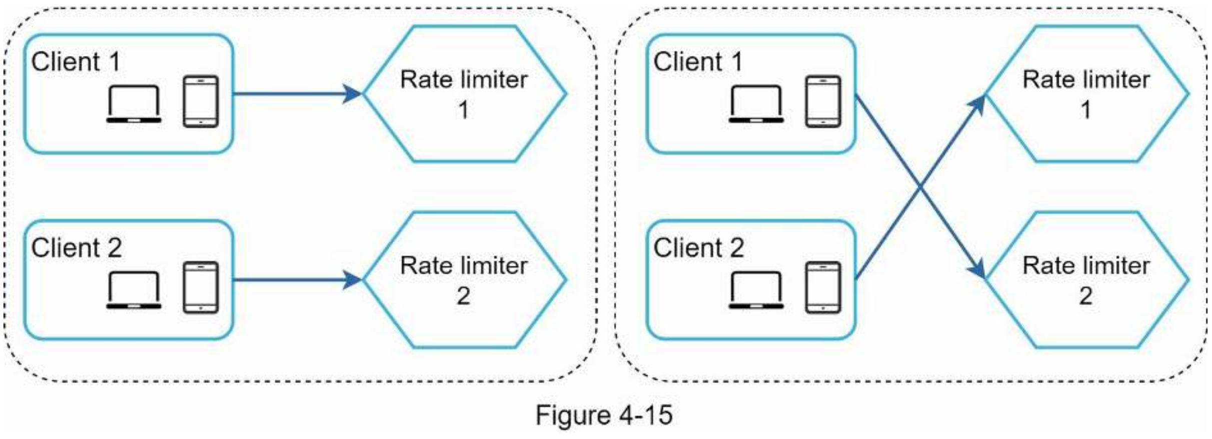


Figure 4-15

One possible solution is to use sticky sessions that allow a client to send traffic to the same rate limiter. This solution is not advisable because it is neither scalable nor flexible. A better approach is to use centralized data stores like Redis. The design is shown in Figure 4-16.

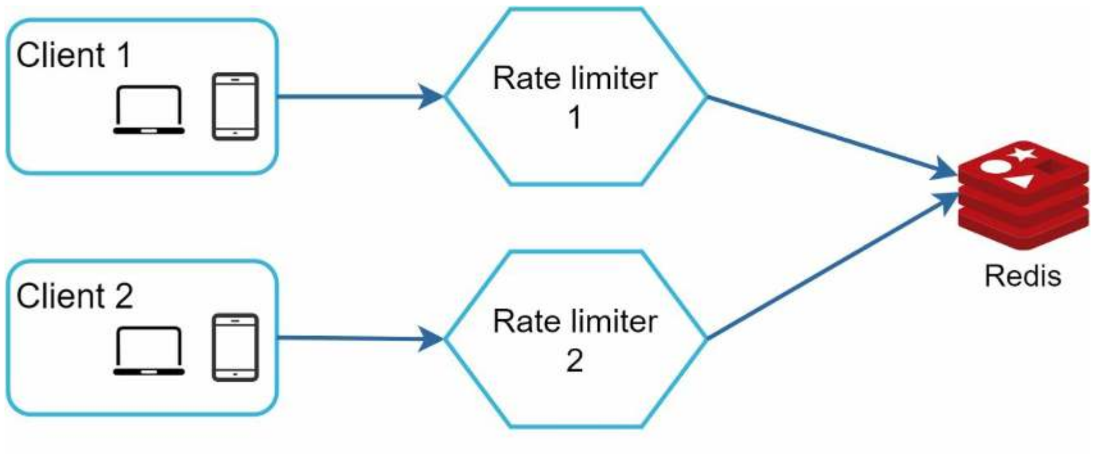


Figure 4-16

Performance optimization

Performance optimization is a common topic in system design interviews. We will cover two areas to improve.

First, multi-data center setup is crucial for a rate limiter because latency is high for users located far away from the data center. Most cloud service providers build many edge server locations around the world. For example, as of 5/20 2020, Cloudflare has 194 geographically distributed edge servers [14]. Traffic is automatically routed to the closest edge server to reduce latency.

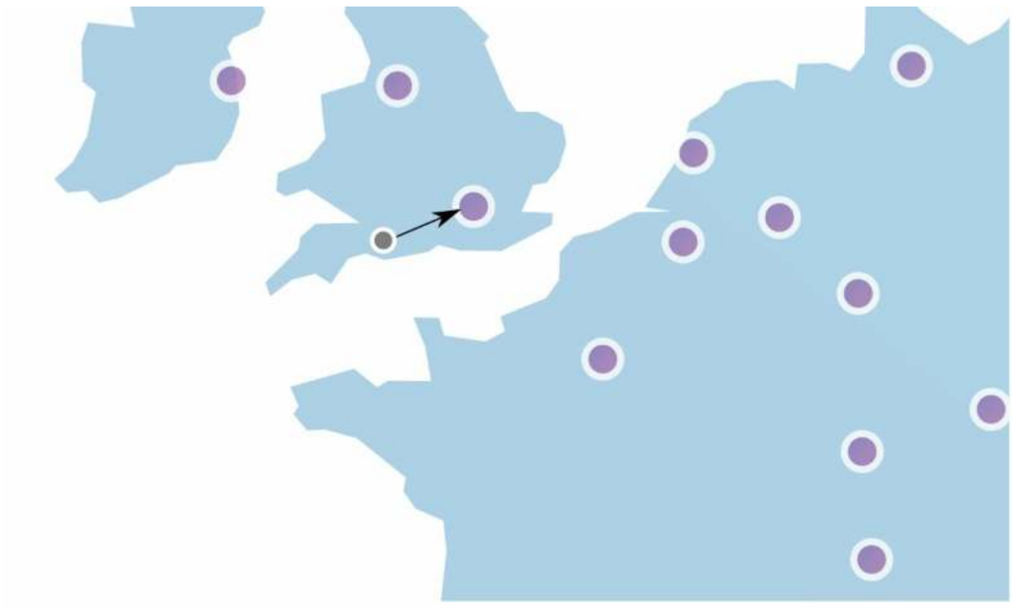


Figure 4-17 (source: [10])

Second, synchronize data with an eventual consistency model. If you are unclear about the eventual consistency model, refer to the “Consistency” section in “Chapter 6: Design a Key-value Store.”

Monitoring

After the rate limiter is put in place, it is important to gather analytics data to check whether the rate limiter is effective. Primarily, we want to make sure:

- The rate limiting algorithm is effective.
- The rate limiting rules are effective.

For example, if rate limiting rules are too strict, many valid requests are dropped. In this case, we want to relax the rules a little bit. In another example, we notice our rate limiter becomes ineffective when there is a sudden increase in traffic like flash sales. In this scenario, we may replace the algorithm to support burst traffic. Token bucket is a good fit here.

Step 4 - Wrap up

In this chapter, we discussed different algorithms of rate limiting and their pros/cons. Algorithms discussed include:

- Token bucket
- Leaking bucket
- Fixed window
- Sliding window log
- Sliding window counter

Then, we discussed the system architecture, rate limiter in a distributed environment, performance optimization and monitoring. Similar to any system design interview questions, there are additional talking points you can mention if time allows:

- Hard vs soft rate limiting.
    - Hard: The number of requests cannot exceed the threshold.
    - Soft: Requests can exceed the threshold for a short period.
- Rate limiting at different levels. In this chapter, we only talked about rate limiting at the application level (HTTP: layer 7). It is possible to apply rate limiting at other layers. For example, you can apply rate limiting by IP addresses using Iptables [15] (IP: layer 3). Note: The Open Systems Interconnection model (OSI model) has 7 layers [16]: Layer 1: Physical layer, Layer 2: Data link layer, Layer 3: Network layer, Layer 4: Transport layer, Layer 5: Session layer, Layer 6: Presentation layer, Layer 7: Application layer.
- Avoid being rate limited. Design your client with best practices:
    - Use client cache to avoid making frequent API calls.
    - Understand the limit and do not send too many requests in a short time frame.
    - Include code to catch exceptions or errors so your client can gracefully recover from exceptions.
    - Add sufficient back off time to retry logic.

Congratulations on getting this far! Now give yourself a pat on the back. Good job!

Reference materials

[1] Rate-limiting strategies and techniques: https://cloud.google.com/solutions/rate-limiting-strategies-techniques

[2] Twitter rate limits: https://developer.twitter.com/en/docs/basics/rate-limits

[3] Google docs usage limits: https://developers.google.com/docs/api/limits

[4] IBM microservices: https://www.ibm.com/cloud/learn/microservices

[5] Throttle API requests for better throughput: https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-request-throttling.html

[6] Stripe rate limiters: https://stripe.com/blog/rate-limiters

[7] Shopify REST Admin API rate limits: https://help.shopify.com/en/api/reference/rest-admin-api-rate-limits

[8] Better Rate Limiting With Redis Sorted Sets: https://engineering.classdojo.com/blog/2015/02/06/rolling-rate-limiter/

[9] System Design — Rate limiter and Data modelling: https://medium.com/@saisandeepmopuri/system-design-rate-limiter-and-data-modelling-9304b0d18250

[10] How we built rate limiting capable of scaling to millions of domains: https://blog.cloudflare.com/counting-things-a-lot-of-different-things/

[11] Redis website: https://redis.io/

[12] Lyft rate limiting: https://github.com/lyft/ratelimit

[13] Scaling your API with rate limiters: https://gist.github.com/ptarjan/e38f45f2dfe601419ca3af937fff574d#request-rate-limiter

[14] What is edge computing: https://www.cloudflare.com/learning/serverless/glossary/what-is-edge-computing/

[15] Rate Limit Requests with Iptables: https://blog.programster.org/rate-limit-requests-with-iptables

[16] OSI model: https://en.wikipedia.org/wiki/OSI_model#Layer_architecture

CHAPTER 5: DESIGN CONSISTENT HASHING

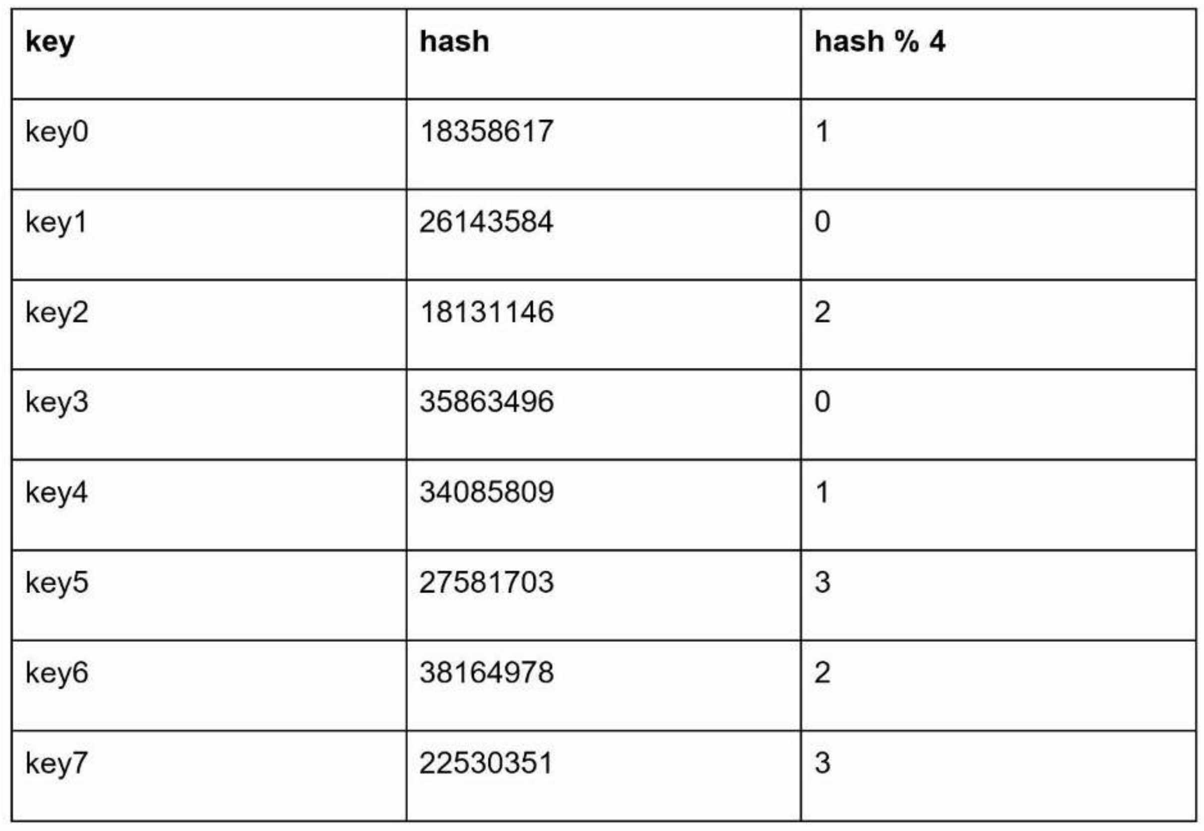
To achieve horizontal scaling, it is important to distribute requests/data efficiently and evenly across servers. Consistent hashing is a commonly used technique to achieve this goal. But first, let us take an in-depth look at the problem.

The rehashing problem

If you have *n* cache servers, a common way to balance the load is to use the following hash method:

serverIndex = hash(key) % N, where *N* is the size of the server pool.

Let us use an example to illustrate how it works. As shown in Table 5-1, we have 4 servers and 8 string keys with their hashes.



key	**hash**	**hash % 4**
key0	18358617	1
key1	26143584	0
key2	18131146	2
key3	35863496	0
key4	34085809	1
key5	27581703	3
key6	38164978	2
key7	22530351	3

Table 5-1

To fetch the server where a key is stored, we perform the modular operation *f(key) % 4*. For instance, *hash(key0) % 4 = 1* means a client must contact server 1 to fetch the cached data. Figure 5-1 shows the distribution of keys based on Table 5-1.

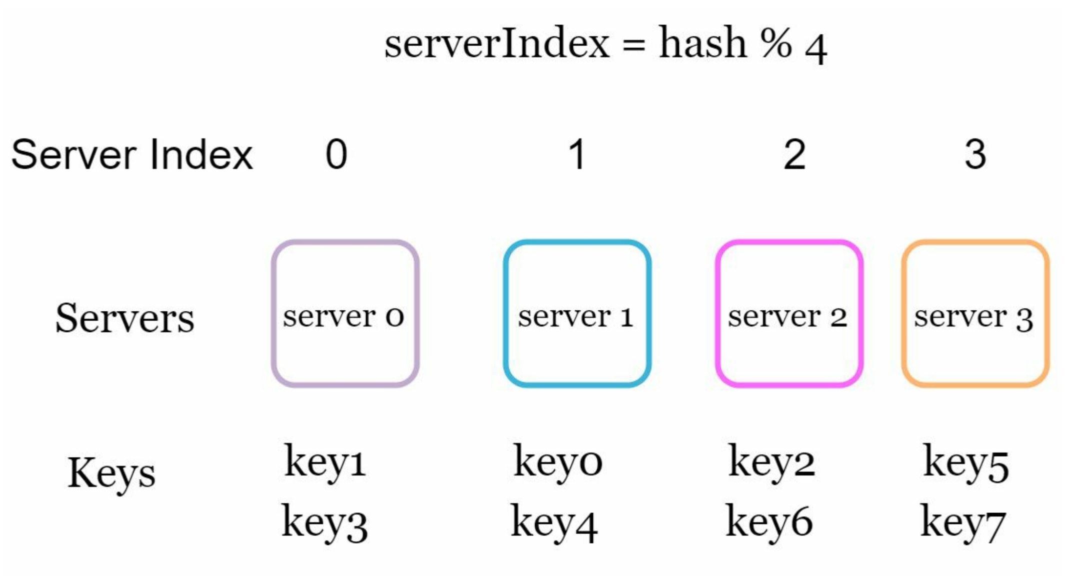
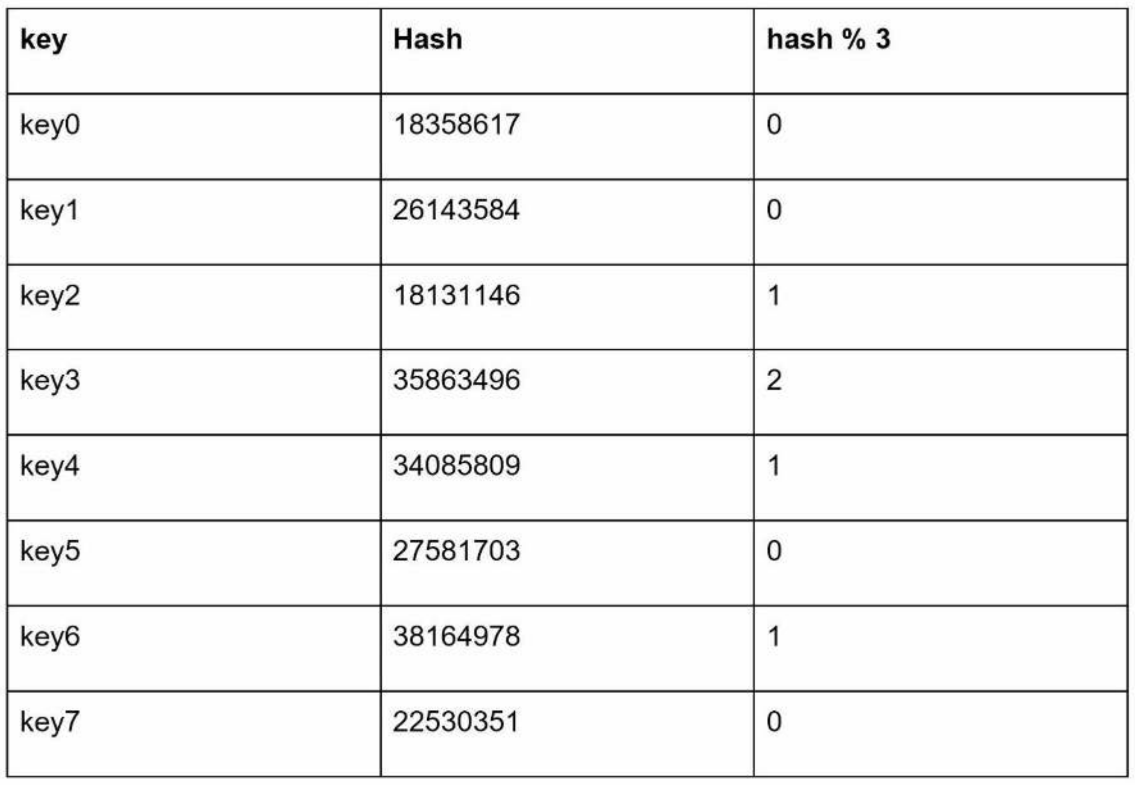


Figure 5-1

This approach works well when the size of the server pool is fixed, and the data distribution is even. However, problems arise when new servers are added, or existing servers are removed. For example, if server 1 goes offline, the size of the server pool becomes 3. Using the same hash function, we get the same hash value for a key. But applying modular operation gives us different server indexes because the number of servers is reduced by 1. We get the results as shown in Table 5-2 by applying *hash % 3*:



key	Hash	hash % 3
key0	18358617	0
key1	26143584	0
key2	18131146	1
key3	35863496	2
key4	34085809	1
key5	27581703	0
key6	38164978	1
key7	22530351	0

Table 5-2

Figure 5-2 shows the new distribution of keys based on Table 5-2.

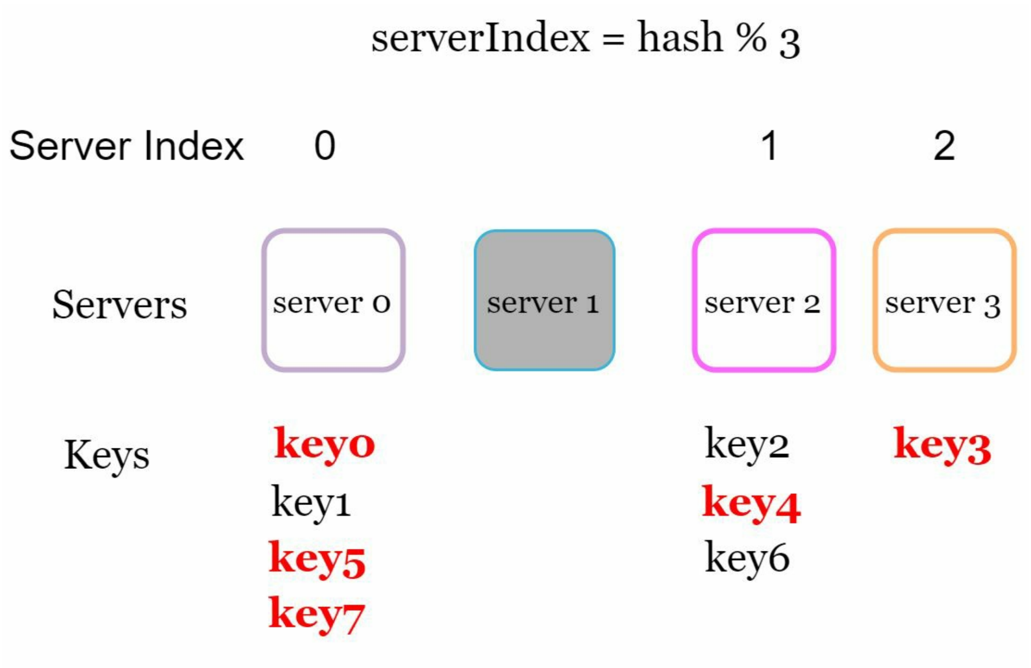


Figure 5-2

As shown in Figure 5-2, most keys are redistributed, not just the ones originally stored in the offline server (server 1). This means that when server 1 goes offline, most cache clients will connect to the wrong servers to fetch data. This causes a storm of cache misses. Consistent hashing is an effective technique to mitigate this problem.

Consistent hashing

Quoted from Wikipedia: "Consistent hashing is a special kind of hashing such that when a hash table is re-sized and consistent hashing is used, only k/n keys need to be remapped on average, where k is the number of keys, and n is the number of slots. In contrast, in most traditional hash tables, a change in the number of array slots causes nearly all keys to be remapped [1]".

Hash space and hash ring

Now we understand the definition of consistent hashing, let us find out how it works. Assume SHA-1 is used as the hash function f, and the output range of the hash function is: *x0, x1, x2, x3, …, xn*. In cryptography, SHA-1's hash space goes from 0 to 2^160 - 1. That means *x0* corresponds to 0, *xn* corresponds to 2^160 – 1, and all the other hash values in the middle fall between 0 and 2^160 - 1. Figure 5-3 shows the hash space.

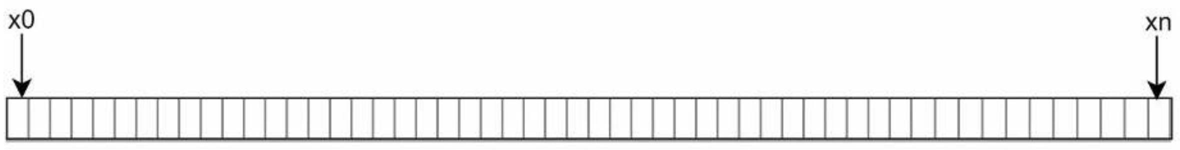


Figure 5-3

By collecting both ends, we get a hash ring as shown in Figure 5-4:

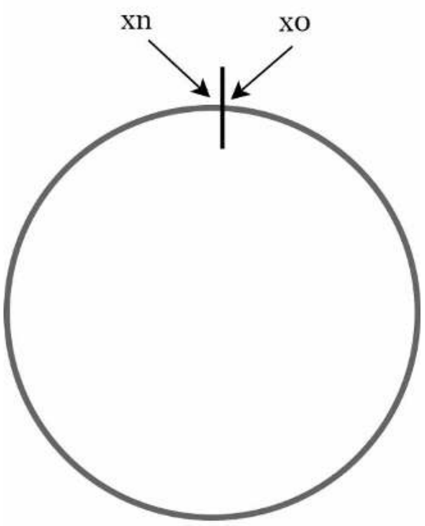


Figure 5-4

Hash servers

Using the same hash function f, we map servers based on server IP or name onto the ring. Figure 5-5 shows that 4 servers are mapped on the hash ring.

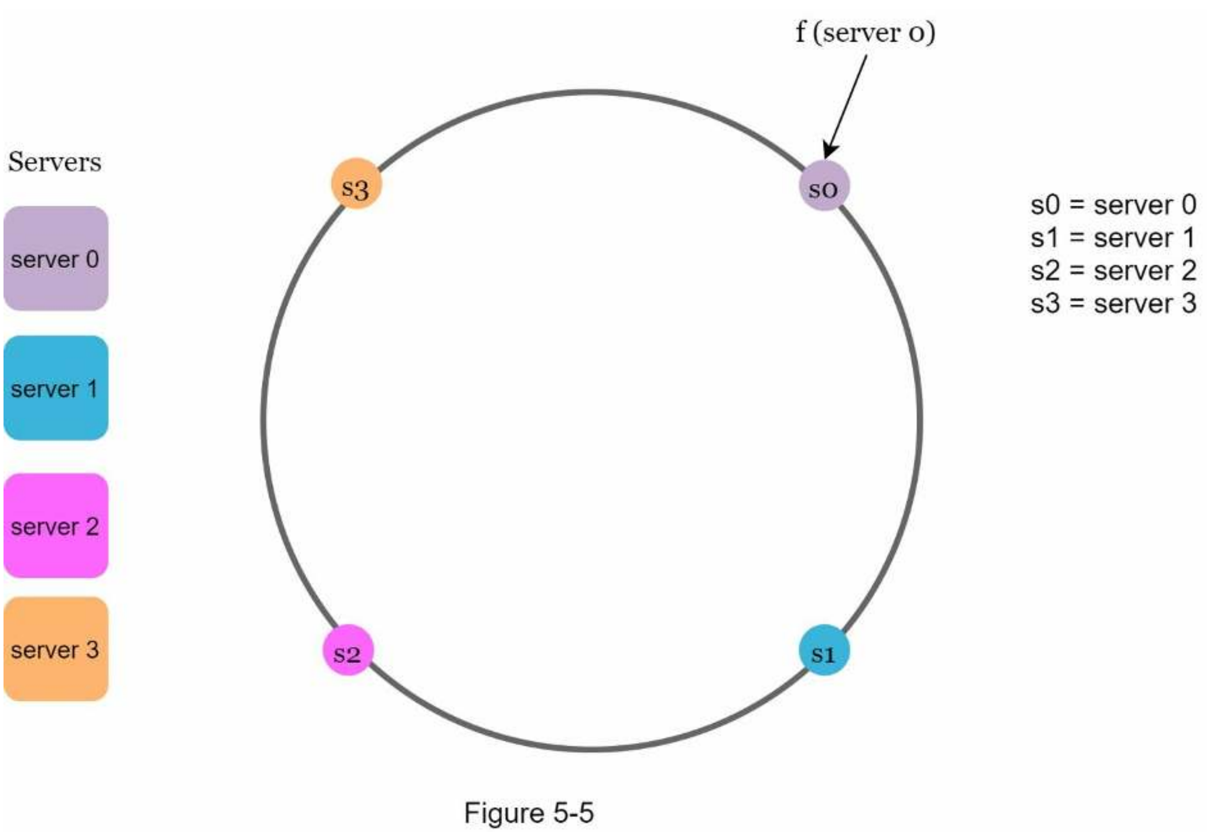


Figure 5-5

Hash keys

One thing worth mentioning is that hash function used here is different from the one in “the rehashing problem,” and there is no modular operation. As shown in Figure 5-6, 4 cache keys (key0, key1, key2, and key3) are hashed onto the hash ring

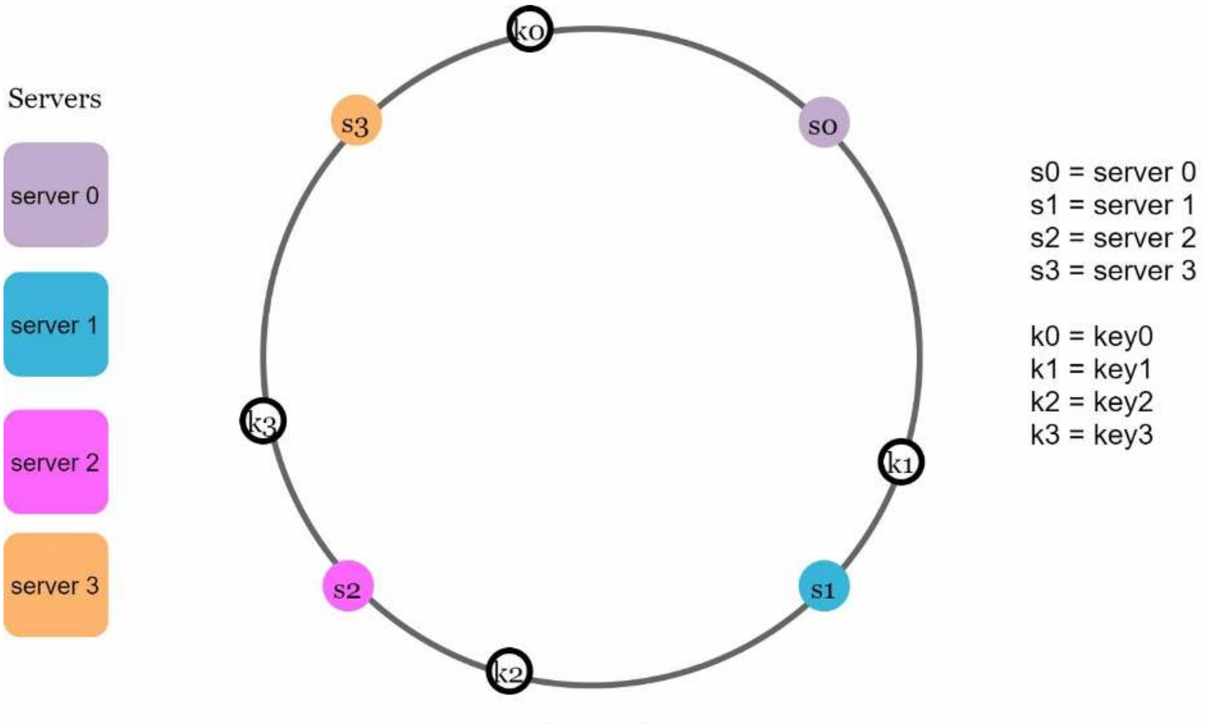


Figure 5-6

Server lookup

To determine which server a key is stored on, we go clockwise from the key position on the ring until a server is found. Figure 5-7 explains this process. Going clockwise, *key0* is stored on *server 0*; *key1* is stored on *server 1*; *key2* is stored on *server 2* and *key3* is stored on *server 3*.

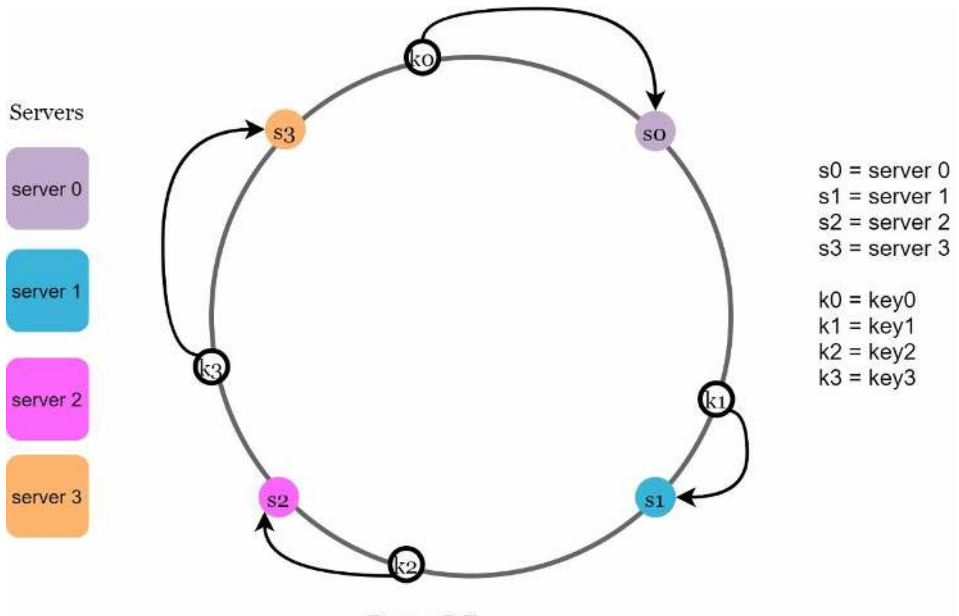


Figure 5-7

Add a server

Using the logic described above, adding a new server will only require redistribution of a fraction of keys.

In Figure 5-8, after a new *server 4* is added, only *key0* needs to be redistributed. *k1*, *k2*, and *k3* remain on the same servers. Let us take a close look at the logic. Before *server 4* is added, *key0* is stored on *server 0*. Now, *key0* will be stored on *server 4* because *server 4* is the first server it encounters by going clockwise from *key0*'s position on the ring. The other keys are not redistributed based on consistent hashing algorithm.

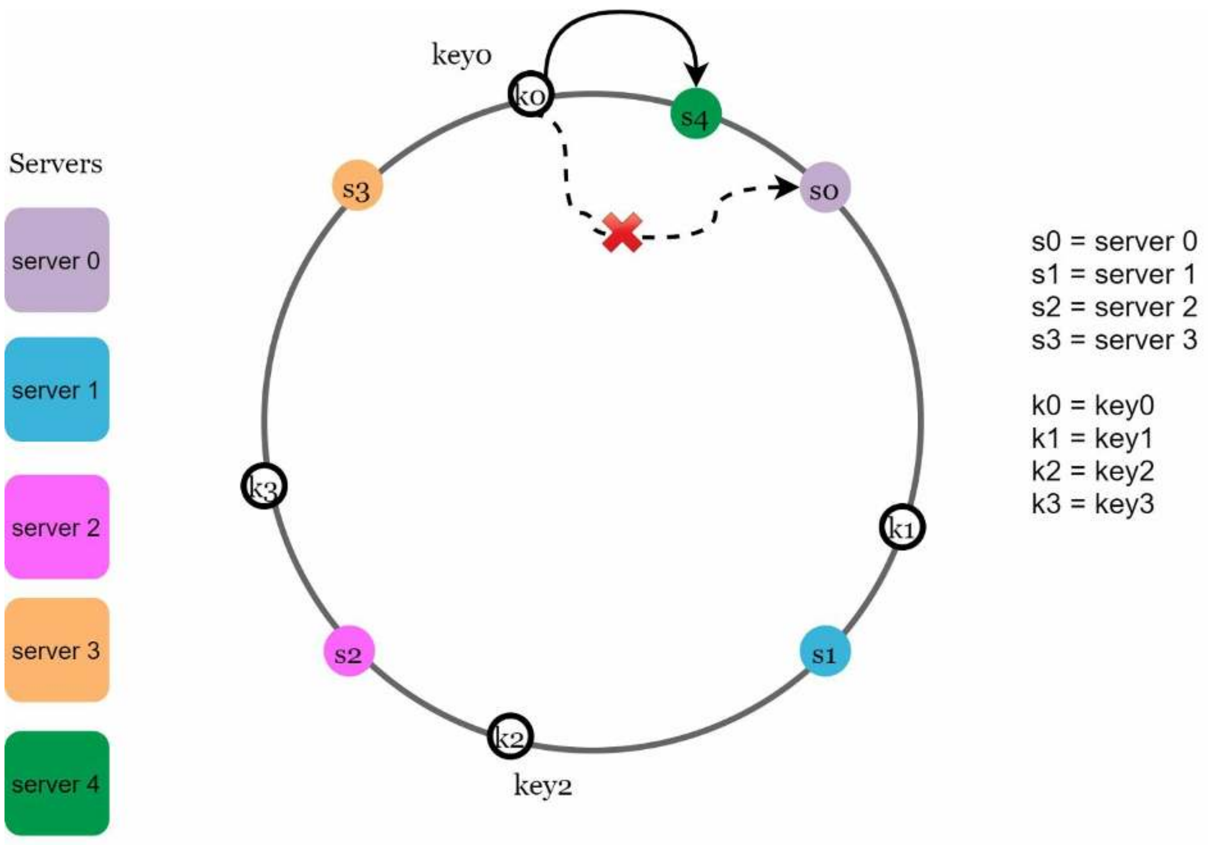


Figure 5-8

Remove a server

When a server is removed, only a small fraction of keys require redistribution with consistent hashing. In Figure 5-9, when *server 1* is removed, only *key1* must be remapped to *server 2*. The rest of the keys are unaffected.

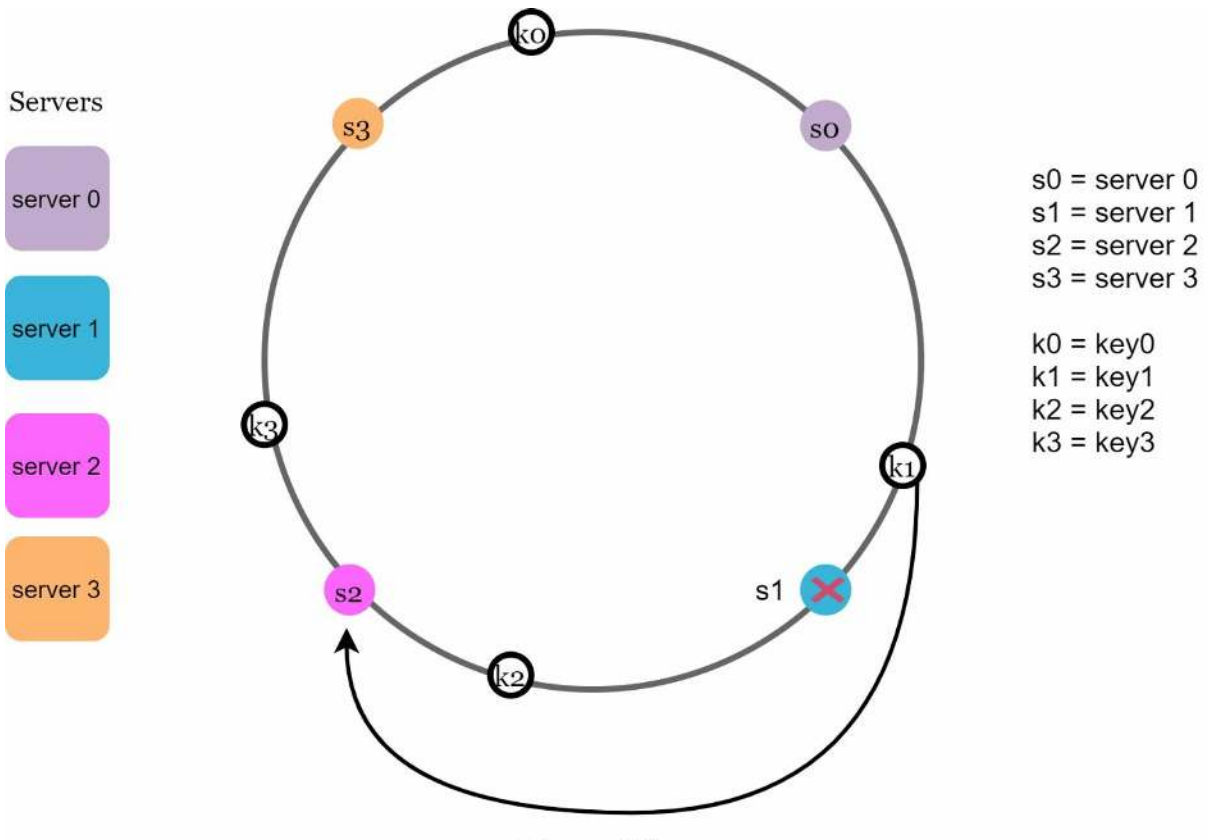


Figure 5-9

Two issues in the basic approach

The consistent hashing algorithm was introduced by Karger et al. at MIT [1]. The basic steps are:

- Map servers and keys on to the ring using a uniformly distributed hash function.
- To find out which server a key is mapped to, go clockwise from the key position until the first server on the ring is found.

Two problems are identified with this approach. First, it is impossible to keep the same size of partitions on the ring for all servers considering a server can be added or removed. A partition is the hash space between adjacent servers. It is possible that the size of the partitions on the ring assigned to each server is very small or fairly large. In Figure 5-10, if *s1* is removed, *s2's* partition (highlighted with the bidirectional arrows) is twice as large as *s0* and *s3's* partition.

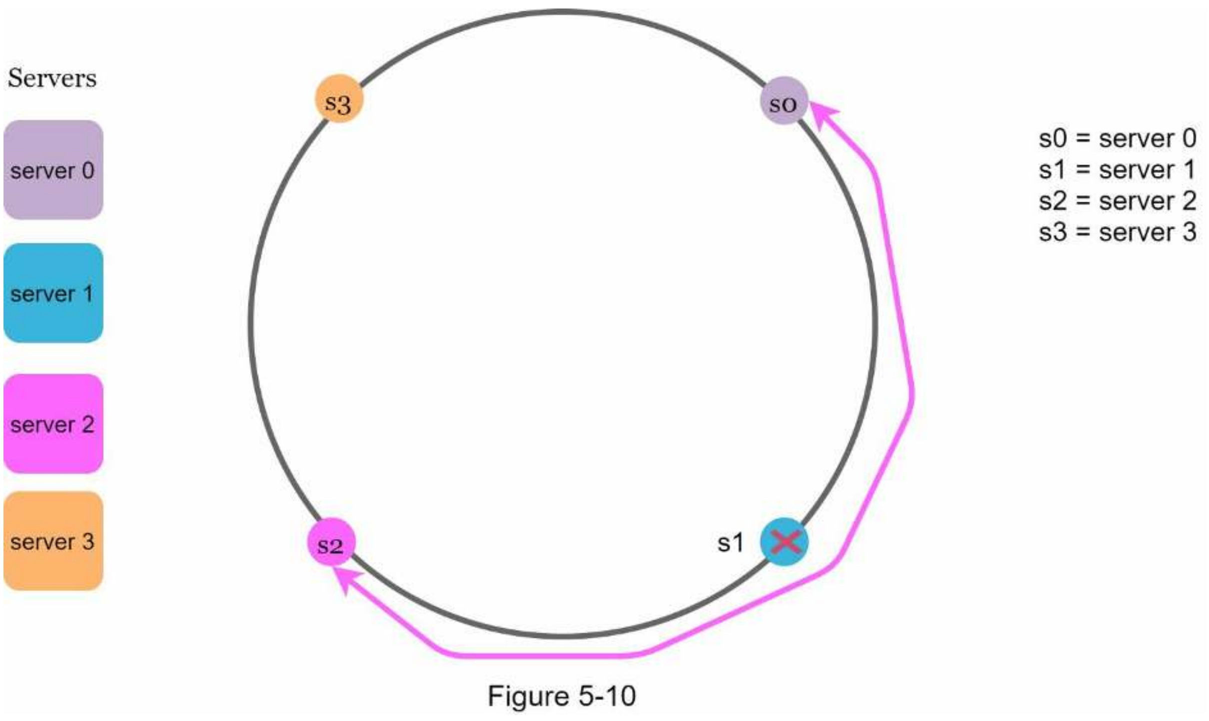


Figure 5-10

Second, it is possible to have a non-uniform key distribution on the ring. For instance, if servers are mapped to positions listed in Figure 5-11, most of the keys are stored on *server 2*. However, *server 1* and *server 3* have no data.

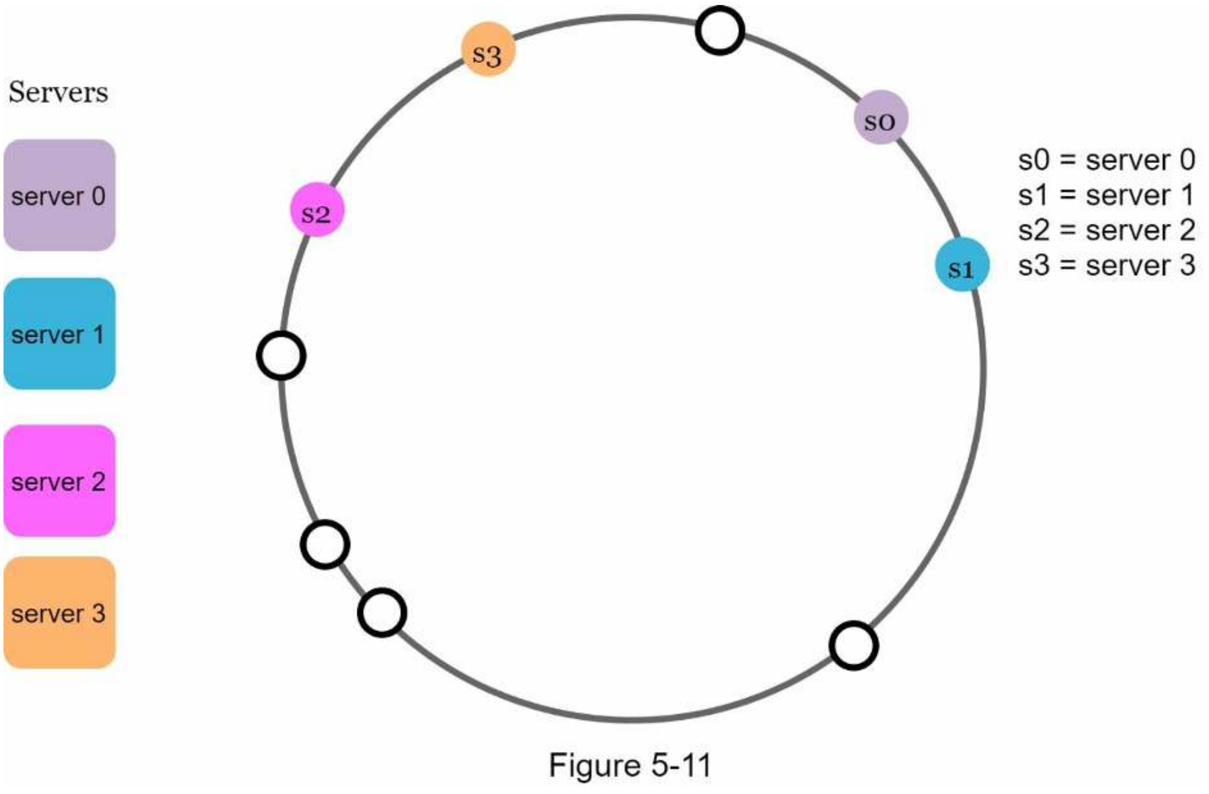


Figure 5-11

A technique called virtual nodes or replicas is used to solve these problems.

Virtual nodes

A virtual node refers to the real node, and each server is represented by multiple virtual nodes on the ring. In Figure 5-12, both *server 0* and *server 1* have 3 virtual nodes. The 3 is