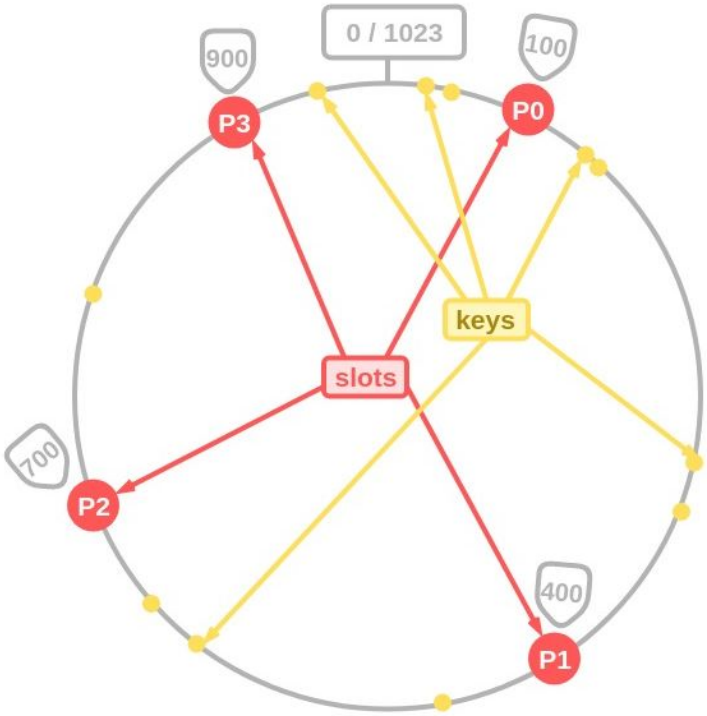
Scalable Web Cache Using Consistent Hashing



BY:
Sarthak Singhal (20171091)
Gaurav Batra(20171114)
Puru Gupta(20171187)

CONTENTS

Web Cache

Modern Problem

A key performance measure for the World Wide Web is the speed with which content is served to users. As traffic on the Web increases, users are faced with increasing delays and failures in data delivery. Two main reasons for these are:

- congested networks
- swamped servers

Date travels slowly through congested networks. Swamped servers (facing more simultaneous requests than their resources can support) will either refuse to serve certain requests or will serve them very slowly. Network congestion and server swamping are common because network and server infrastructure expansions have not kept pace with the tremendous growth in Internet use. Servers and networks can become swamped unexpectedly and without any prior notice. For example, a site mentioned as the “trending” on the evening news may have to deal with a ten thousand fold increase in traffic during the next day.

Solution? Caching

Caching has been employed to improve the efficiency and reliability of data delivery over the Internet. A nearby cache can serve a page(if cached) quickly even if the originating server is swamped or the network path to it is congested. Besides widespread use of caches also engenders a general good: if requests are intercepted by nearby caches, then fewer go to the source server, reducing load on the server and network traffic to benefit all users.

Drawbacks of using a single cache

Providing a group of users, specially on a network with a single shared caching machine has several drawbacks:

- If this caching machine fails, all users are cut off from the cache and all the users again directly request the server
- While running, a single cache is limited in the number of users it can serve
- It may become a bottleneck during periods of intense use
- Due to limited storage, the cache will suffer ”false misses” when requests are repeated for objects which it was forced to evict for lack of space

Multiple cache usage

To achieve fault tolerance, scalability, and aggregation of larger numbers of requests several solutions are proposed using systems of several cooperating caches. Advantages of using multiple cache servers are:

- Failure of one or more servers, doesn’t affect the functioning of others and they can serve request normally
- Multiple caches can handle more client request due to increased resources
- Chances of false misses decreases

Orthodox Hashing

Approach

One of our desired goals when using multiple cache servers is that data should be distributed equally among all cache servers to prevent a particular server becoming a bottleneck for the entire system. This is why we use hash functions as they tend to distribute their inputs randomly among all possible locations. It is obvious that cache servers will come up and go down over time. This dynamic number of running cache servers makes normal hash schemes difficult to use. A normal hash scheme is generally given by:

$$Server.Index = H(x)\%N \text{ where } N = number\ of\ running\ servers$$

Rehashing

Now if we want to use this scheme, we need to tackle one problem. If the servers keep on coming up and going down, N keeps on changing and under such a scheme, essentially every URL needs to be rehashed to a new cache. If this is not done:

- We would keep getting a cache miss as the data would be present in different cache and we would be searching it in some other cache
- This would place heavy load on the original server and thus the purpose of introducing cache would be defeated
- This would also use up more cache space as multiple copies would be stored on different cache servers

Considering the above problems, it is necessary to rehash. Rehashing might be a herculean task because it will either require a scheduled system downtime to update mappings or create read replicas of the existing system which can service queries during the migration. In other words, a lot of pain and time wastage.

Drawbacks

Even rehashing might not be able to solve our problem in a distributed setting. The problem in a distributed system with simple rehashing is that there is state stored on each node; a small change in the cluster size for example, could result in a huge amount of work to reshuffle all the data around the cluster. As the cluster size grows, this becomes unsustainable as the amount of work required for each hash change grows linearly with cluster size.

The problem is further exacerbated by asynchronous information propagation through the internet. At any one time, different clients will have different information about what caches are up or down(referred to as client's view). At any time, many different views will pervade the system which has potential drawbacks:

- If each view causes a URL to map to a different cache, we will soon find that each URL is stored in all caches
- Furthermore, with these multiple views in place it becomes difficult to argue that all caches will receive the same amount of load

Consistent Hashing

Abstract

All these problems can be handled by using a distribution scheme that would not depend directly upon the number of cache servers. Consistent hashing is one such scheme that facilitates the distribution of data across a set of nodes in such a way that minimizes the re-mapping of data when nodes are added or removed. It is a distributed hashing scheme that operates independently of the number of servers or objects in a distributed hash table by assigning them a position on an abstract circle, or hash ring. This allows servers and objects to scale without affecting the overall system. It does away with all inter-cache communication, yet allows caches to behave together in one coherent system. Some advantages of consistent hashing over other cache schemes are:

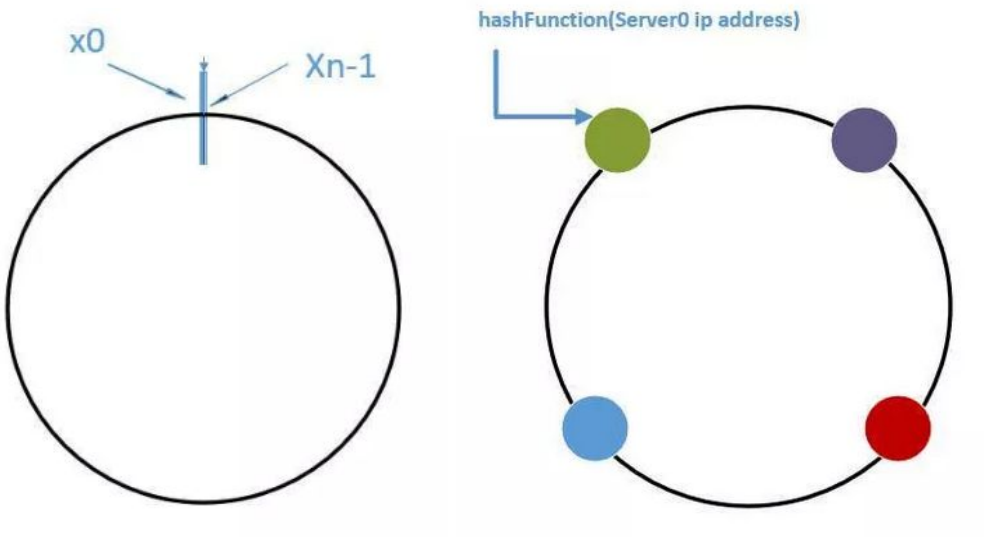
- Since all clients contact the same cache for a given page, the caching system suffers only one miss per page, regardless of the number of cooperating caches, rather than one miss per cache per page.
- Miss rate is further reduced because we do not make redundant copies of a page in different caches (considering clients have same view)
- This also leaves more space for other pages to be kept in the cache and increases chances of hit.

Setup

The setup for consistent hashing is as follows:

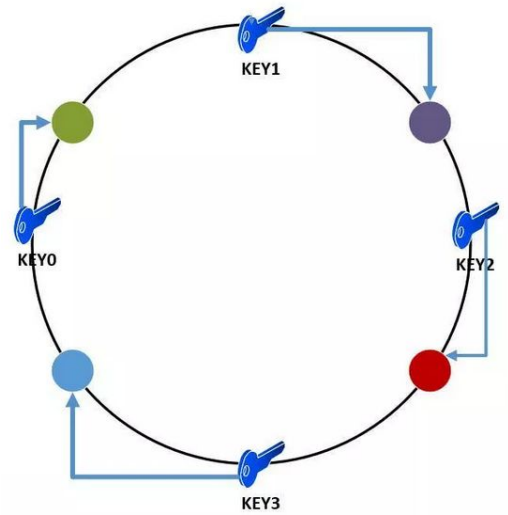
- Creating Hash Key Space:- Consider we have a hash function that generates 32 bit integer hash values. We can represent this as an array of integers with 2^32 slots. We'll call the first slot x_0 and the last slot x_{n-1}.
- Representing the HashSpace as a Ring:- Imagine that these generated integers are placed on a ring such that the last value wraps around.

- Placing cache servers on the hash ring:- Using our hash function, we map each cache server to a specific place on the ring.



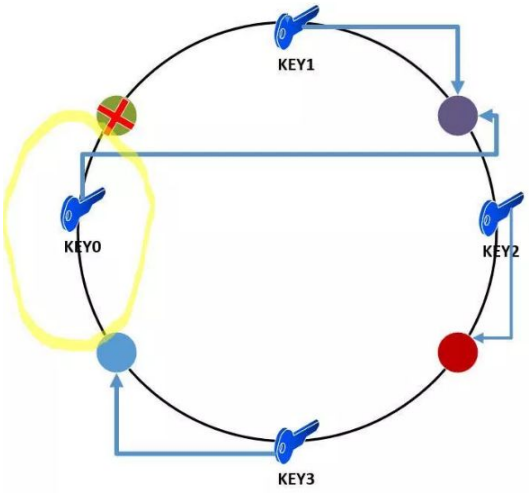
Determining cache server for data points

Using our hash function, we map each data-point/key to a specific place on the ring. After mapping the data on the hashring, we move in any one direction: clockwise or anti-clockwise and search for the nearest cache server on the ring. Then we place the data on the first server that we find. In our example, it looks like below figure:



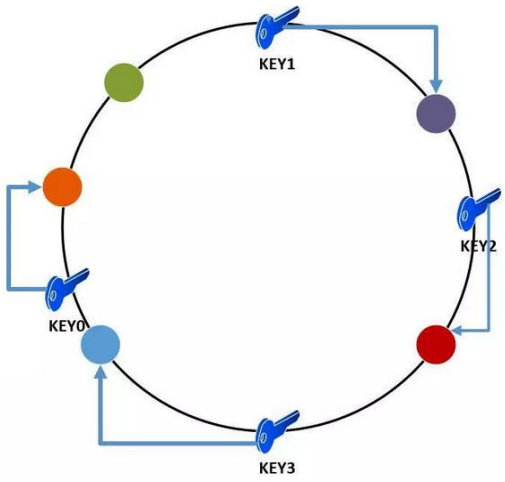
Removing a cache server

A server might go down in production but consistent hashing ensures it has minimal effect on our cache system. Let's say the green server goes down. This results in data belonging to this server being distributed between blue and purple servers. Whereas data belonging to blue and purple remains untouched. This is because when green went down, we needed to search for another closest server in the chosen direction which resulted in redistribution of server's data. However if a value was mapped to any other server, even after the green server went down, they were the nearest server to the value on the hashring and thus no redistribution took place.



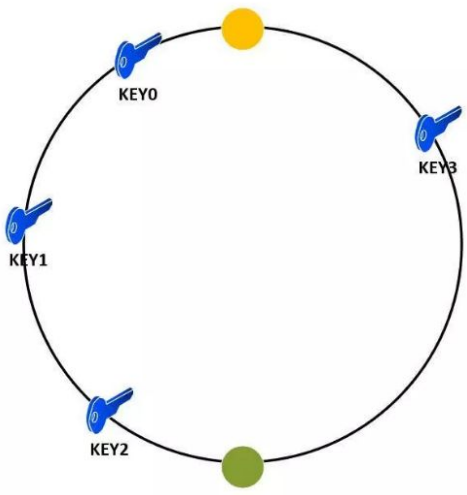
Adding a cache server

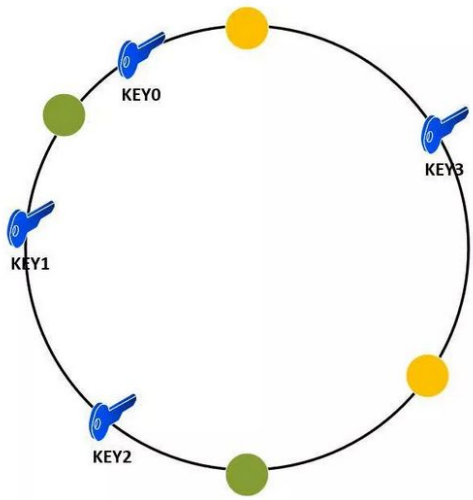
A server may again come up after going down in production but consistent hashing ensures it has minimal effect on our cache system. Let's say we are adding another server and it gets mapped between the blue and green server. All the data points lying only between these two servers need to be remapped as only for them the nearest server can change. Specifically, keys between orange and blue servers need to be remapped. On an average we need to redistribute only K/N data points where K is the total number of data points and N is the total number of servers.



Server Replication (Virtual Node)

There might be a case that if few servers are there, all data points may hash between any two servers resulting in overloading of that server. To ensure even distribution of data among cache servers, we assign not one but many locations to a server on the hashring. Number of copies depends on the situation and may even be different for each server. Say if server S_1 was twice as powerful as other servers, we could make twice the number of copies of S_1. As the number of replicas in the ring increases, load on each server becomes more and more uniform. In real systems, the number of replicas is very large (>100).





Handling multiple views

Consider a system with m caching machines and c clients, each with a view of an arbitrary set of half the caching machines. There exists a theorem stating that if O(logm) copies of each caching machine are made and the copies and URLs are mapped to the ring using a good basic hash function, then the following properties hold:

- Balance:- In any one view, URLs are distributed uniformly over the caching machines in the view.
- Load:- Over all the views, no machine gets more than O(logc) times the average number of URLs.
- Speed:- No URL is stored in more than O(logc) caches.

Code

Abstract

We have implemented a miniature version of consistent hashing as well as simple hashing that are used to cache the requests between the Server and Client. Later is implemented to make a comparison between the two schemes. Also for simplicity we have assumed that all clients have the same view of the system, i.e. we have assumed synchronous information propagation through the internet. The Code has 4 major components:

1. Client
2. Server
3. Hashing Scheme
4. Cache server

Directory structure of the submission is as follows:

```
├──── Report.pdf        report
├──── Video.txt         drive link of the video
├──── Client.py         code for the client
├──── Ring.py           code for consistent hashing and cache server
├──── Server_2.py       code for server for normal hashing
├──── Server.py         code for server for consistent hashing
└──── SimpleHash.py     code for normal hashing
```

Code Components

1. Client

   Client.py contains a basic implementation of a client that has a number of functions to interact with the server.

   - Add/remove cache server
   - Add/remove data

- Check cache server state
- Check performance of system

2. Server

   We have implemented two servers (Server.py, Server_2.py), one for each hash scheme. The server can handle the requests from multiple clients and uses TCP/IP protocol to establish connection with the clients. Several functions are implemented to handle the requests made by the clients.

3. Hashing Scheme

   - Consistent Hashing:- Ring.py contains a ConsistentHashedRing class that creates a ring object which has a number of nodes on it. Each node has a number of replicas (virtual nodes) on the ring where each node is a cache to store the data. Sortedlist() is used to implement the ring (a list which keeps elements in sorted order).
   - Simple Hashing:- SimpleHash.py contains code for simple hash functions that implements orthodox hashing. In this configuration server ID is the server number.

4. Cache Server

   Ring.py also contains a node class which defines the cache server. Each cache is a LRU cache which is implemented using OrderedDict() which stores elements in order of their timestamp. Each cache node has a fixed size (say 50 elements) and we can access/remove elements in a way to ensure that only most recent elements are there in the cache.

Functions

We have implemented the following functions for functioning of our cache system:

- stats : To know the status of running caches.
  Syntax: stats

- perf : To know the performance of caches(mainly load sharing).
  Syntax: perf
- clean : To reset the server for analysing performance.
  Syntax: clean <replica> <cache_size> [CH]
  Syntax: clean [NH]
- test : To add multiple nodes and data points to test the cache.
  Syntax: test <no. of nodes> <no. of data points>
- addnode : To add a cache server to the system.
  Syntax: addnode <server:ip> [CH]
  Syntax addnode [NH]
- rmnode : To delete a node from the system.
  Syntax: rmnode <key>
- get : To get data by key.
  Syntax: get <key>
- add : To add data to a cache.
  Syntax: add <key> <value>

Performance Analysis

Metrics

We have used the following performance metrics:

1. Hit ratio:- This is calculated by first adding data in the cache servers and then accessing some of the data randomly.
2. Time:- Time to add servers and data is compared. This is used mainly between the hashing schemes.
3. Load sharing:- We also tested how the load is distributed among various cache servers. For this we computed:
    - Mean data entries in servers
    - Standard deviation of data entries in servers
    - Standard deviation as percentage of mean

    Low value of standard deviation as percentage of mean tells that load is distributed uniformly among the servers.

Setup

The following parameters are varied while measuring performance:

VARIABLE	DEFAULT VALUE
Hash scheme	Consistent hash
Number of server replicas	50
Cache size	50
Number of cache servers	3
Number of data entries	10,000

To overcome any issues related to the number of replicas in consistent hashing, while making any comparison between consistent and orthodox hashing, we take 50 times the number of cache servers/cache size in orthodox hash scheme than in the consistent hash scheme (As we are making 50 replicas for each cache server in consistent hashing).

Analysis

Each of the above parameters were varied and performance was measured. The results are as follows:

1. Hash scheme

   To test our consistent hash scheme, we compared it with orthodox hashing scheme. For this along with consistent hashing we also implemented distributed cache servers using orthodox hashing schemes. For the analysis, LRU cache size limit was set high enough that it could hold the entire data, so nothing would get evicted. This is because, we want to test cache hits of the consistent hash algorithm, rather than limits of each cache server. If we use smaller cache size, then the hit ratio would reduce as data would keep on getting removed as new data is added.

   a. Orthodox hashing with rehashing

      - Since cache limit is larger than data added, with rehashing we would get 100% *hit ratio* in orthodox hashing scheme. Consistent hash scheme would also give 100% cache hit.
      - Only difference would be in *time* as rehashing needs to remap every data point again to a new server.
        - To test this we first initialised by adding 3 servers and 10,00,000 data points.
        - We noted the time to add a server.
        - We also noted the time of redistribution when a server went down or came up.

- The results are:

ACTION	CONSISTENT HASH	ORTHODOX HASH
Server up	0.008835s	0.689311s
Server down	0.006552s	0.026676s

- *Load distribution* would be equal as hashing maps each server/data to a random point on the hash ring and we have used sufficient servers to ensure uniform distribution.

b. Orthodox hashing without rehashing

- *Hit ratio* would vary largely as in orthodox hash scheme the same URL would be mapped/searched in different servers due to changing number of running servers.
  - First we commented the rehashing part in the code.
  - To test this, we initialised both the schemes by adding 3 servers and 1000 data entries.
  - Then we noted the hit ratio for various scenarios
  - Consistent hashing does redistribution so the hit ratio is always 100%. On the other hand orthodox hashing gives a very low hit ratio when the number of servers is changed once. When it is changed again, it gives almost 0 hit ratio.
  - Up -> server coming up
  - Down -> server going down

ACTION	CONSISTENT HASH	ORTHODOX HASH
Initial	100%	100%
up	100%	26.8%
down	100%	37.2%

up -> down	100%	0%
down -> up	100%	0%

- *Time* would also be similar as rehashing is omitted. Consistent hashing may take slightly more time as it redistributes some data.
- *Load distribution* would be uniform in both the cases as sufficient number of servers are used.

2. Number of server replicas
    - *Hit ratio* would increase as cache memory would increase (trivial).
    - *Time* would increase as each time we add a cache server, more replicas are needed to be created (trivial).
    - Major advantage of creating a large number of replicas is that it helps in uniform *load distribution* among the servers. We want the ring to contain as many servers/virtual nodes as possible to maintain uniform distribution.
        - Following results are with 5 servers and 100 data points.
        - SD as % M decreases as we increase the number of replicas which indicates distribution becoming more and more uniform.

REPLICAS	**MEAN**	**SD**	**SD as % of M**
1	20	17.87	89.37%
10	33.33	10.96	32.9%
100	33.33	6.80	20.42%c

3. Cache size
    - Increase in cache size would increase the *hit ratio* (trivial).

CACHE SIZE	HIT RATIO
10	14.28%
50	55.14%
70	70.54%

- No significant effect on *time*. Data would be evicted at much less rate if cache size is increased.
- No effect on *load distribution* as it depends on how the nodes are located on the ring.

4. Number of cache servers
    - Increase in number of servers would result in increased *hit ratio* (trivial).
    - No significant effect on *time*. Just more time to add the servers.
    - Major advantage is that it leads to a somewhat better *load distribution*. But since we are using a large number of replicas, it doesn't affect much. If a low number of replicas are made, then it will affect to a great extent. We can exchange the number of replicas and number of servers to get similar results as obtained above.
5. Number of data entries
    - Increase in data points results in low *hit ratio* (trivial).
    - No effect on *time*. Just more time to add more data.
    - Few data points may cause non uniform *load distribution* due to points being mapped between two servers on the ring.

Number of cache servers and data points do not affect much as compared to other factors

References

- https://www.acodersjourney.com/system-design-interview-consistent-hashing
- http://www.cs.cmu.edu/~srini/15-744/S02/readings/K+99.html
- https://www.toptal.com/big-data/consistent-hashing
- https://www.ably.io/blog/implementing-efficient-consistent-hashing
- https://www.scs.stanford.edu/14au-cs244b/labs/projects/le_stephenson_manandhar.pdf