Web Caching with Consistent Hashing

Shreya Ravi
sravi2@stanford.edu

Yu Qing (Ivan) Zhou
ivanz@stanford.edu

Yu Liu
yul231@stanford.edu

Julius Zhang
juliuszh@stanford.edu

Abstract

A scable web caching system needs to be robust to the dynamic changes of the cache server nodes while ensuring high cache hit rates. This paper presents an implementation of the consistent hashing algorithm based on a hash ring to address issues of high cache miss rates resulting from adding and removing cache server nodes in a naive implementation of distributed key hashing. Experiments with simulated internet traffic on the Stanford myth cluster demonstrates that our implementation supports flexible removal and creation of cache server nodes and achieves high cache hit rates compared to a benchmark naive implementation of distributed caching.

1. Introduction

Caching web content improves the responsiveness internet access, reduces network congestion, and optimizes data delivery over the Internet. By storing web resources in the cache, one can use distributed resources to handle future requests. A good implementation of web caching supports fault tolerance, scalability, and high hit rates in order to serve a large number of requests.

In this project, we implement a scalable web caching mechanism using consistent hashing. We distribute web cache evenly among a set of cache servers and adopt the consistent hashing technique to allow for changes in the number of the cache servers in response to flexible user demands. Section 2 is a literature review of previous work on web caching. Section 3 covers both the high level design and key components in our system architecture. In section 4, we give quantitative evaluation of our system and compare it with the traditional hashing table. Finally, in section 5, we present suggestions for the future work.

2. Previous Work

Implementing efficient web caching is key to ensuring reliable internet content delivery [1]. One approach is to use dedicated cache servers to store cached content, so that the master server forwards requests to a specific cache server based on the hashed value of the request. However, with this approach, a naive implementation, in which one uses the hashed request value modulo the number of cache servers to select the cache server, is ineffective because it does not take into account the overhead of removing and adding cache server nodes, resulting in high cache miss rates. One remedy is to use consistent hashing as proposed in [2] and [3], which is based on the implementation of a hash ring.

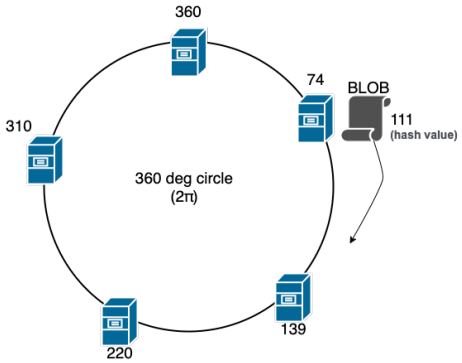

Figure 1. Illustration of a hash ring. [4]

In consistent hashing, cache servers are conceptually positioned along a hash ring of buckets, as shown

in Figure 1, where the numbers above the cache servers represent the bucket numbers in the hash ring. The master server then forwards a request to the cache server that is closest clockwise to the bucket corresponding to the hashed value of the request. The consistent hashing with a hash ring implementation is much more efficient and fault tolerant than a naive implementation of web caching, because when a server is removed or added, only the requests going to cache server next to the added or removed server clockwise are affected. In practice, the performance can be improved even more by the use of virtual nodes [5] which ensures that cache servers are distributed more evenly on the hash ring, thereby ensuring lower cache miss rates in expectation.

3. Architecture

In this section, we are going to go through our system architecture. 3.1 talks about the high-level system overview. 3.2 explains the consistent hashing and heartbeats implementation in the master server.

3.1. System Overview

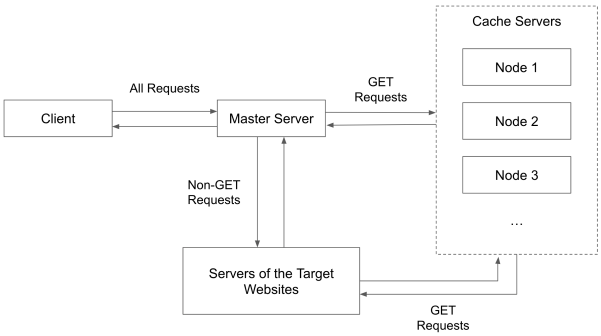

Figure 2. The system design

Figure 2 shows our system design. There are two primary components. First, there is a master server that listens to requests from our client, acting as a proxy server. Second, there are a set of cached servers that store cached web content in a distributed manner. The cached servers communicate with the master server through constant heartbeats to report their active status.

Once receives a GET request from a client, the master server will send it to one of the active cache server. If the cache server contains a cached copy for the request, it will reply with the cached content. Otherwise, it will query new content from the web server that corresponds to the request, return the fetched content back to the master server, and meanwhile store a local copy of the content in its cache. The master server will finally send the content back to the client who initiates the request.

The client will be agnostic to whether the content comes from the cached copy or fresh from the target server.

For all the non-GET requests, the master server will directly send to the servers of the target websites, without going through any cached server.

3.2. Master

There are two critical components in the design of the master server: the consistent hashing algorithm that is used to assfign a GET request to one of the cache servers based on the URL; heartbeats that are used to monitor the status of cache servers and maintain a list of active cache servers.

3.2.1 Consistent Hashing

The master server maintains a hash ring, as shown in 1. When a new cache server is added to the system, it gets registered by the master server. To distribute the workload of cache servers more evenly, we create 100 virtual nodes per cache server. The 100 virtual nodes are then added to the hash ring, with locations determined by the hash value of the public IP address of the cached server and the ID of the virtual node. We use MD5 message-digest algorithm [6] as the hashing function for consistent hashing. It is selected based on its low computation requirement.

When a GET request is received, the master server hashes the URL in the GET request and locates the first virtual node that is closest to the hashed position clockwise. The virtual node will then be mapped to the physical cache server and the master server will forward the GET request to that cache server.

3.2.2 Heartbeats

Cache servers send regular heartbeat messages to communicate with the master server to stay alive. On a high level, a cache server sends heartbeats to the master server at a pre-determined interval. On the mas-

ter server side, the master server calls a flush method periodically to remove cache nodes whose heartbeat messages have not been received for a given amount of time.

In our implementation, a heartbeat message is implemented as an encoded plaintext message for simplicity, and there is a custom designed data structure for storing metadata for each node. An example is as follows.

```
node_meta = {
    "hostname": hostname,
    "instance": instance,
    "nodename": nodename,
    "port": port,
    "vnodes":
    number of virtual nodes,
    "lastHeartbeat":
    time of last heartbeat message,
}
```

The node metadata structures are stored in the hash ring and benchmark single hash table implementations. The key attributes should be self-explanatory. We use 'vnodes' upon node initialization to determine the number of virtual nodes needed for a node creation as per [5]. When the master server receives a heartbeat message, it updates the 'lastHeartbeat' field in the correspondent node metadata structure. Periodically, the master server calls the flush method to mark inactive servers (whose heartbeat message has not been received for a specified amount of time) as dead and remove them from the hash ring, or the single hash table implementation in the benchmark. For our design choice, we reduce communication overhead on the server side since the server does not respond to each heartbeat message. The flush method is needed for both correctness and efficiency, as straggler nodes can become a performance bottleneck and in even worse situations cause deadlock. For example, a node for a hashed key may never respond due to network partitions or server errors. The periodic flushing provides a simple interface that ensures liveness by removing faulty and straggling nodes.

3.3. Cache Servers

Each cache stores the response from origin servers in memory. GET requests are forwarded from the master server to the single cache server that would hold the data for that GET request, as determined by the consistent hashing algorithm. If the cache does not hold the response from that GET request, it forwards that request to the origin server, stores the response in its in-memory map, and sends the response back to the master server. If the cache is forwarded that GET request again by the master server, it simply searches its in-memory map for the response and sends that response to the master server. Thus, the cache servers work to serve the client its response faster, reduce network traffic, and decrease load for the origin servers.

3.3.1 In-Memory Map

The in-memory map is implemented as a Python dictionary, in which each key is the request and each value is an object storing both the response that should be forwarded back to the master server and a timestamp for when the cache serve got the response from the origin server.

This in-memory map is used to provide a response much faster than forwarding the request to the origin server would. The map consists key-value pairs to store the requests and responses (and associated metadata), and lookups are $O(1)$, making them highly efficient. In-memory lookups are several orders of magnitude faster than network speeds, even within the same data center[7]. However, as the cache grows in size, it will not be able to fit all its data in memory and must page data out to disk. Reading from disk is significantly slower than reading from memory, although if the origin server is far away, we still improve performance significantly by using the cache server that reads from disk.

To both maintain correctness and improve performance, we introduce timestamps as metadata stored alongside the HTTP responses to expire cache entries and flush them out periodically.

3.3.2 Expiration

The purpose of expiring cache entries is to maintain correctness of the responses and improve performance of the cache.

Imagine there exists a GET request x to an origin server, and the corresponding response y is stored in a cache server. Let's say that the origin server changes

its response to GET request x to response z. Then, if the client sends GET request x again to its proxy (the master server), it will get back the outdated response y rather than the new response z because the cache still holds y. This correctness issue can be solved by forcing cache entries to expire after a given amount of time (e.g. 1 day) after the response was first fetched from the origin server.

This expiration is enforced by only serving non-expired cache entries to the client, and when a cache server receives a GET request with a cache entry whose response is expired, the cache will send the request to the origin server and update its cache entry with the new response. This maintains correctness, and, by periodically flushing the cache of expired entries, we can reduce the cache size and keep more of the cache in memory, improving performance by reducing disk reads which are slower.

3.3.3 Multi-threading

Although a single threaded implementation is both correct and performant, a multi-threaded implementation can improve performance of serving GET requests by hiding network I/O latency. The main challenge is making sure the shared data structure, the in-memory map, does not serve as a limiting bottleneck while also not introducing data races. Thus, we have developed the following design.

We use two types of locks: one coarse-grained lock used to synchronize accesses to the map and one fine-grained lock per cache entry used to synchronize accesses to each cache entry. Both types of locks are single-writer, multiple-reader locks (SWMR). Each incoming request from the master server is handled by its own thread, which finishes execution once it sends its response to the master server.

When serving a cache hit, the thread acquires the coarse-grained lock as a reader and acquires the associated fine-grained lock (stored in the map as metadata associated with the request) as a reader. Thus, if the client makes the same GET request several times in quick succession, all of these requests (which are cache hits) can be serviced simultaneously. When servicing a cache miss, the thread acquires the coarse-grained lock as a writer and adds the GET request as a key with an empty response to the app. Then, the thread acquires the fine-grained lock as a writer and demotes its writer coarse-grained lock to a reader lock. Then, after receiving the response from the origin server, the thread writes the response data to the cache entry and releases its locks. A separate thread is also running to periodically flush the cache. This thread attains the coarse-grained lock as a writer to remove entries from the cache.

For this implementation, it is important that the SWMR locks are fair and don't starve any threads so that all requests can be served in a timely manner.

4. Evaluation

4.1. Setup

We have implemented our own web server to display large web page contents, which serves as the origin server as mentioned in the design scheme. We set up one master server with four cache servers to start. The way we used to interact with the master server is to send multiple GET requests to query HTTP contents from the master server. The test workflow was shown below. For each step, we kept track of the cache miss rate (measured in percentage) and the total time to complete the step, where one hundred of requests were sent.

1. Fetch all the contents. The purpose of this step was to fetch all the contents from the origin server and save in our cache servers. Since all cache servers were newly added, we were supposed to get 0% cache hits for both configurations. The time in both configurations should be similar as the only operation is to fetch contents from the origin server.

2. Fetch all again. We were trying to show the general benefit of using cache in this step. This should produce 100% cache hits and faster fetching process in both configurations, as the contents being fetched were saved in cache servers from the previous step. The time in this step should be smaller than that of step 1 in both configurations, since fetching from cache memory should be faster than fetching from the origin servers, no matter what hashing scheme we used here.

3. Add a new cache server to the system. This step is to show the benefit of using consistent hash-

ing when adding a new cache to the system compared with traditional hashing. This should produce 80% of cache hits for consistent hashing, and a much lower cache hits for traditional hashing.

4. Remove the first added cache server, then fetch again. This is to test the functionality of our system when one of the cache servers crashed. This should produce a much greater cache hits for consistent hashing compared with that of traditional hashing, as all keys need to be rehashed in the other configuration.

4.2. Data

The text contents we used to display on our web server is from `https://www.gutenberg.org` [8] . We divided the contents into sections. We have tried section size from 1kB to 10MB, but there was no great change in latency test results. Hence, we used 1kB as the section size. The parameter at the end of each test URL is used to specify the number of section to fetch. Once the web server got the requests from cache servers, it would send back the one section of contents from the last fetch point.

4.3. Baseline

For the baseline, we implemented a single hash table server that naively stores the cache server nodes and forwards the request to the specific cache server based on the hashed request value modulo the number of cache servers. This is the naive approach for implementing a cache server suggested by [3]. We chose this implementation because it is particularly easy to implement and illustrates the central problem that consistent hashing attempts to solve, i.e. removing and adding nodes affects the assignment of cache servers.

The benchmark was set to send the exactly the same requests to a simple proxy, the only functionality of which is to forward all the requests to the origin server. The time to fetch one hundred requests in the benchmark is 0.161s. The baseline was set to be using the traditional single hash table.

4.4. Results

The key metrics we use for evaluation in this section are the cache hits rates and total time to fetch 100 requests for the each test step.

Step	Consistent Hashing	Traditional Hashing
1	0%	0%
2	100%	100%
3	80%	19%
4	83%	23%

Table 1. Cache hit rates to fetch 100 requests with consistent hashing turned on and off.

Table 1 shows the cache hit rates. In step 1, we got 0% cache hit and 100% cache miss for both configurations as expected. The cache hit rates of adding and removing cache servers (step 4 and 5) for consistent hashing are 80% and 83%. On the contrary, the cache hit rates for traditional hashing are only 19% and 23%. The cache hit rate using our consistent hashing is about three times higher than that of traditional hashing case. In traditional hashing, the hashing scheme is highly dependent on the number of cache servers. Therefore, when adding or removing cache servers from the system, all the keys would be changed, increasing the load on the origin server. This can be solved by consistent hashing as the hashing scheme operated independently of the number of cache servers in the hash table by assigning them a position on the hash ring.

Step	Consistent Hashing	Traditional Hashing
1	24.902s	25.003s
2	24.856s	24.967s
3	19.911s	19.978s
4	25.034s	24.428s

Table 2. Total time to fetch 100 requests with consistent hashing turned on and off.

Table 2 shows the total time used to complete each step in both configurations. The time for step 1 is much higher compare with that of benchmark test. Compared the time in the first two steps for both configurations, there is only a slight improvement on the fetching speed. Fetching contents from master and cache server did introduce latency to some extent. Also, given the fact that all the servers were running on myth machines, the time used to fetch from origin server might be similar to the time used to fetch from our master server.

5. Future work

There are a variety of possible directions in both algorithms and the implementation worthy of possible

exploration. One direction is to investigate implementing LRU cache with consistent hashing [9]. Our current implementation of consistent hashing imposes an upper limit on the number of cache servers that can be added to the hash ring. It is interesting to design data structures and algorithms that support consistent hashing with an arbitrary number of cache servers. Another approach focused more on applications is to use ideas from consistent hashing to implement distributed hash tables (see [10] and [11]). On the security side, it is also interesting to examine applications built on top of consistent hashing in the context of DDoS attacks [12].

In our current evaluation, the experiments are run on the myth cluster with simulated internet traffic. For future work, it is more illuminating to evaluate the performance of consistent hashing on real network with real internet requests. It is interesting how deployment of consistent hashing servers across different geolocations affects performance.

6. Conclusion

In conclusion, our scalable web caching system using consistent hashing is able to handle repeated access to the same web content in response to elastic user demands. Our design consists of two major parts, a master server and a group of cache servers. The master server performs consistent hashing and forwards requests to corresponding cache servers. Each cache server spawns off threads to process the request from the master server and fetch contents from its in-memory map or the origin server. The explain and heartbeat logic in cache servers help maintains their correctness.

We evaluated the performance of our web caching system by logging the cache hit and total time to process one hundred requests when adding and removing cache servers compared with traditional hashing system. The results has shown that the cache rates of our system is about three times higher than that of traditional hashing system when the number of cache servers changed. The improvement of time is not noticeable in this case given the tests ran on myth machines with simulated traffic. For future optimization, we can investigate LRU algorithm, distributed hash tables, security defense, and deployment across different geolocations.

References

[1] Jia Wang. A survey of web caching schemes for the internet. *SIGCOMM Comput. Commun. Rev.*, 29(5):36–46, oct 1999.

[2] David Karger, Alex Sherman, Andy Berkheimer, Bill Bogstad, Rizwan Dhanidina, Ken Iwamoto, Brian Kim, Luke Matkins, and Yoav Yerushalmi. Web caching with consistent hashing. *Computer Networks*, 31(11):1203–1213, 1999.

[3] Tim Roughgarden and Gregory Valiant. Cs168: The modern algorithmic toolbox lecture 1 ... - stanford university.

[4] Wikimedia Commons. File:consistent hashing sample illustration.png — wikimedia commons, the free media repository, 2021. [Online; accessed 1-June-2022].

[5] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, feb 2003.

[6] Ronald Rivest. The md5 message-digest algorithm. Technical report, 1992.

[7] Jeff Dean. Designs, lessons and advice from building large distributed systems, Oct 2009.

[8] Project gutenberg. [Online; accessed 1-June-2022].

[9] Kaiyi Ji, Guocong Quan, and Jian Tan. Asymptotic miss ratio of lru caching with consistent hashing. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 450–458, 2018.

[10] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, page 53–65, Berlin, Heidelberg, 2002. Springer-Verlag.

[11] Hao Zhang, Yonggang Wen, Haiyong Xie, and Nenghai Yu. A survey on distributed hash table (dht): Theory, platforms, and applications. 2013.

[12] Baruch Awerbuch and Christian Scheideler. A denial-of-service resistant dht. In Andrzej Pelc, editor, *Distributed Computing*, pages 33–47, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.