Implementing a Web Cache using Consistent Hashing

Elmer Le, Abhay Manandhar, Rob Stephenson
CS244b, Fall 2014

Abstract

This paper describes a proxy server that provides a distributed, scalable web cache, implemented by utilizing consistent hashing. The goal of this project was to prototype a system that could cache web content, while also being resilient to the addition or removal of cache nodes by keeping the number of cache misses low.

1. Introduction

Traditional hashing is typically done with a fixed number of buckets. While the technique is simple, the disadvantage is that with the change in the number of buckets, every element must be re-hashed and assigned to the appropriate bucket. This is not a huge deal for single-process systems or data structures like hash tables, where the cost of re-hashing is amortized over many faster operations. However, for hash tables that operate over distributed systems, we must account for nodes joining and leaving the system, so a traditional hashing scheme would cause far too many re-hashes to be practical. In the specific case of a web cache, this would translate to having a large number of cache misses.

In our system, we use a consistent hashing scheme to minimize the number of cache misses when nodes leave and join the network. Each node is responsible for caching a range of hash values. We built our system with the initial assumption that each node would be roughly equal in terms of hardware specifications and performance, so each node is responsible for an equal-sized portion of the range of possible hash values. These portions are disjoint and cover all possible hash values. Ultimately, this allows the number of servers caching content to grow over time, with minimal disruption to content that is already cached.

2. Architecture

From the client's point of view, our web cache acts as a proxy server for a web browser. The client points their browser at a fixed host, which then handles packet forwarding and caching logic. There are two types of servers in our system: a single master server, which serves as the proxy server, and a set of cache servers. The master server maintains a list of the active cache servers. Each second, every cache server sends a heartbeat message to the master server to announce that it is still alive and functioning; this is also how the master server updates its list of active cache servers. If it hears a heartbeat from a server not in the active list, it treats the new heartbeat as a new cache server and adds it accordingly. Content that has already been cached that now belongs to the newly added cache server is transferred from old cache servers to this new cache server. If a cache server does not send a heartbeat for a pre-set timeout period (currently set to three seconds), then the master server treats that server as dead and removes it from the active list.

The steps involved in an HTTP request from the client are as follows:

1. The client sends the request to the master server.
2. The master server checks if the request is a GET/HEAD.

GET/HEAD request

3. The master server determines which cache server should hold the cached content based on the requested URI and forwards this request to that cache server.
4. The cache server checks if that URI content is already cached. If so, it just returns its cached content to the master server. If not, it will query the origin server for the content, cache it, and then return the content.
5. The master server forwards the content back to the client.

POST request

3. The master server forwards the packet to the origin server.
4. When the response comes back, the master server forwards the response to the client.

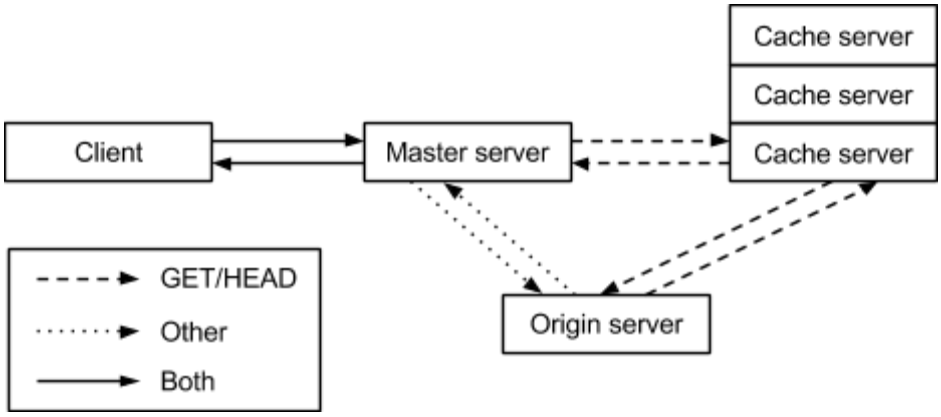

Figure 1: Overview of the system architecture.

In our current implementation, the master server is a single process that we use to determine cache content distribution and hashing. Thus, it is a single point of failure. However, for the sake of demonstrating the concept of consistent hashing, we assume that it will not fail. In a real deployment, we could improve our system by using multiple master nodes to ensure that the master server is also fault tolerant.

Each cache server uses an in-memory LRU cache to hold cached contents. The LRU cache is set to hold a maximum total amount of content size (default currently set to 100 MB), so the number of cached resources is not relevant so long as their total size is under the maximum. This maximum can be easily configured, but one limitation is that it is currently set at compile-time, so we do not have the ability to test with different LRU cache sizes.

Our servers use two different methods to communicate. For RPC calls, we used the XDRPP module; this supports all of our communication between the master and cache servers, and between the cache servers. For all other requests that come from or are going to an external server, we wrote our own HTTP client to handle HTTP requests. This is how we receive requests from web browsers on the master server, and how the cache and master servers request content from origin servers.

3. Consistent Hashing Implementation

Our consistent hashing implementation uses an MD5 hash, which is a relatively fast and effective hash function. (Even though MD5 has been shown to have weaknesses as a cryptographic tool, we are only using it to divide up our cache contents randomly and unpredictably, which it should be fine for.) As of now, each cache server is assigned four virtual nodes, each of which is assigned a random point in the MD5 range.

Upon receiving a request at the master server, we hash the requested URI and find which virtual node's point occurs first as we move down the range of hash values. The cache server that virtual node belongs to is then responsible for that URI's content.

When a cache server can no longer be part of the network for whatever reason (hard drive failure, network errors, natural disasters, etc), the master server will no longer receive its heartbeats, so after the timeout period, that server will no longer get any requests for content, and all requests that would have gone to it are assigned to another node instead. Notably, since we are implementing a web cache and not a persistent data storage system, missing content in a cache does not lead to incorrect functionality, so we do not replicate cache contents. This allows us to simplify the system by letting the request fall through to a cache miss when a node dies.

One extra feature that our system supports is to transfer cached data between cache servers when a new cache server is added to the system. When the master server hears a new heartbeat, it assigns virtual nodes to the cache server in the MD5 output range, and then sends a message to cache servers that may be caching content that belong to the new cache server. The old cache servers then iterate through their cached content and check which of its elements are to be assigned to the new cache server, and transfers that content directly to the new cache server. With the feature enabled, we can decrease the amount of cache misses, since the hash values that switch cache servers will not lead to cache misses. However, the tradeoff is an increase in network usage whenever a server joins the network. Furthermore, if we transfer cached content that is never requested again in the future, those packets are wasted. (Notably, we cannot implement this behavior for when nodes are removed from the system, because we have no guarantee if the removed server is still alive.) For the purpose of testing, we turned this feature off when counting cache hit and miss percentages.

We also set a compile-time flag to optionally turn off our consistent hashing calculation. Instead, the master server will use the traditional hashing approach, where the hash value is simply modded by the number of cache servers, and a cache server is chosen accordingly. This allows us to compare the effectiveness of our consistent hashing against a baseline without consistent hashing.

4. Evaluation

4.1 Consistent hashing

In addition to the server types described above, we also implemented a separate statistics server that accumulates statistics on performance. Currently, we are logging the rate of cache hits, rate of cache misses, and when cache servers join the system. The stats server is written in Python, and accepts simple REST HTTP requests to accumulate performance metrics.

We have implemented two ways of interacting with the proxy server. First is to configure a web browser setting and point it to the proxy server, and then then the browser can be used to visit web pages. We also implemented a shell program that can query HTTP contents from the master server to provide a command line alternative to using a web browser. The shell can also take a text file of links and query each of those URLs sequentially.

One metric we used to evaluate our cache's performance was using this shell and a list of links. We compiled a list of 100 images from imgur, ranging from a few kilobytes to tens of megabytes in size. We set up the master server with four cache servers to start. Then, we ran the following steps, once with consistent hashing turned off, and once more with consistent hashing turned on. For each step, in each configuration, we tracked the cache hit percentage and the raw time to complete the step.

1. Fetch all the images. This should produce 0% cache hits in both configurations.
2. Fetch all again. This should produce 100% cache hits in both configurations.
3. Add a new cache server to the system, and then fetch again. This should produce ~80% cache hits with consistent hashing, and ~25% cache hits without consistent hashing.
4. Remove the added node, then fetch again. This should produce 100% cache hits in both configurations due to the high LRU cache size, so all the old content should still be in the original cache servers.
5. Remove another node, then fetch again. This should produce ~75% cache hits with consistent hashing, and ~33% cache hits without consistent hashing.

Step	With Consistent Hashing	Without Consistent Hashing
1	0%, 25.13s	0%, 25.00s
2	100%, 4.36s	100%, 4.38s
3	79%, 5.51s	17%, 24.61s
4	100%, 4.40s	100%, 4.45s
5	64%, 14.03s	46%, 14.21s

Table 1: Cache hit ratio and total time to fetch 100 imgur images with consistent hashing turned on and off.

Note that for both of these configurations, the LRU caches' size limits were set to 100 MB, which was high enough that it could hold the entire imgur contents in memory, so nothing would ever get evicted. The reason for this decision was that we wanted to test the percentage of cache hits and misses of the consistent hashing algorithm, rather than the limits of each individual cache server. If we were to use a substantially smaller cache size, the fact that we use an LRU eviction policy and access the same set of image links sequentially would mean that we would always get cache misses. (By the time the last image has been accessed, the first image would have been already evicted.)

4.2 Proxy performance

To evaluate the raw performance of our proxy server, we measured the time taken to download one small and one large file using wget. We measured 10 download times for the small file and 5 download times for the large file and present the mean download times in the table below.

Small file used: Putty (http://the.earth.li/~sgtatham/putty/latest/x86/putty.exe, 495616 bytes)
Large file used: VLC (http://downloads.sourceforge.net/project/vlc/2.1.4/win64/vlc-2.1.4-win64.exe, 25056116 bytes)

	Not using proxy server	Proxy server with cache miss	Proxy server with cache hit
Small file	2.334 seconds	2.902 seconds	0.084 seconds
Large file	52.098 seconds	57.666 seconds	6.836 seconds

Table 2: Time taken to download a small and large file without using the proxy server, using the proxy server without the content in cache, and using the proxy server with the content already in cache.

We saw a performance overhead of 24.3% and 10.7% for the small and large file respectively when the content was not already in one of the cache servers. When the files were already cached by one of the cache servers, the file downloads took only 4.6% and 13.1% percent of the total time taken to download the small and large files respectively without using the proxy server.

5. Ongoing Work

The core consistent hashing functionality part of our project is working correctly. However, as a web proxy, there are some features that prevent some websites from loading properly. Because the master and cache servers need to be able to request content from origin servers, they need to be able to forward requests from the browser and read HTTP responses. Unfortunately, our own HTTP client isn't able to do this perfectly yet. It does not handle 'chunked' responses that do not contain a 'Content-Length' field in the header, but we are working on it for the project demo.

The proxy server is currently implemented as a single thread of execution, so all requests are processed serially. This causes any large file requests to block the proxy server from processing other requests.

We are currently working on enhancing the proxy server to make use of multiple threads to process requests to the server so a large file request does not block the entire system.

Another feature that could improve the web cache is to rethink the assumption that cache server capacities are fixed and known ahead of time. Hardware varies widely, so we could allow each cache server to specify how much content it can reasonably be responsible for when it is added to the system. Similarly, if web traffic begins skewing towards content on a particular cache server, we could shrink that server's hash range to transfer some of its load to other less popular servers.

6. Conclusion

Our web cache shows a noticeable improvement for repeated access to the same web content, both functioning as a basic web cache alone and as a distributed cache that smoothly supports scaling upward and downward. By using consistent hashing to distribute web content among cache servers, we are able to minimize the number of cache misses that occur when nodes join and leave the system. Additionally, since our master server is configured to act as a web proxy as well, we can demonstrate its effectiveness using real web content examples.

In terms of being fit for actual use, we experimented with a feature that would transfer cache content when servers are added to reduce the amount of cache misses even further. However, there are a few features missing that would allow it to handle the load and variability of a real-world environment. Our single master server needs to be upgraded to support many clients requesting content simultaneously, and our HTTP client needs to be able to interact with a wider range of HTTP request and response formats.