COMP 480/580 — Probabilistic Algorithms and Data Structure Feb 4, 2020

Lecture 7

Lecturer: Anshumali Shrivastava *Scribe By: Jonathan Cai*

1 Introduction: Yahoo! Video Platform

1.1 Background

The Yahoo! Video Platform owns over 20,000,000 video assets, and must service 30,000,000 requests per day, for over 800,000 unique videos, which creates a low ratio of total requests to unique requests. To add to this, video files are large, and a typical server can only hold 500 unique videos in memory and 100,000 unique videos on disk.

The low ratio of total/unique requests combined with large video files makes it difficult to achieve high percentage of hits, avoiding cache misses.

The goal of this problem is to efficiently route requests to machines that cache videos.

1.2 Requirements

Requirements to this problem are as follows:

1. **Stateless Addressing**: Servers are added and deleted and content should automatically route to the sever id given the video filename
2. **Efficiency**: Request Router must recalculate the destination server on every request
3. **Heterogeneous pool of servers**: Server capacities are different. Each sever i has capacity c_i and the load must be balanced in proportion to the loads of the server

2 Last Lecture: Consistent Hashing

As discussed previously, consistent hashing is a viable option, but meets all requirements except the third. There are some issues that need to be addressed with consistent hashing:

2.1 Issue #1: Domino Effect

The Domino Effect is a result of how requests are mapped to servers, specifically how you hash a request, and then traverse the circular address space counter-clockwise until you reach a server. When one server crashes, then all of its load will move to the next server, which is then also likely to crash as a result, and so on and so forth, crashing servers in sequence like dominoes.

2.2 Issue #2: No Proportional Distribution

This is in reference to the third requirement, where you have a heterogeneous pool of servers. The basic consistent hashing algorithm simply allocates a single hash address, which we assume is random uniform across the entire hash range. Thus, all servers, regardless of capacity, are expected to have the same load, or requests routed to it. Therefore, the load is not balanced for servers proportional to their capacity.

2.2.1 Work-around

However, there is a viable work-around to this issue. You can modify consistent hashing to solve this by creating multiple copies of server ID's, proportional to their capacity, and hash them to address spaces across the hash range. For example, if we have three servers, where capacity c is $c_0 = 2$, $c_1 = 5$, $c_2 = 10$, you could make 2 copies of server 0, 5 of server 1, and 10 of server 2. With this solution, we can expect the load of each server to be proportional to its capacity now.

3 Proposed Solution: SPOCA

Standing for A Stateless, Proportional, Optimally-Consistent Addressing Algorithm, this solution, proposed in Chawla et al., avoids the issues above [1].

3.1 The Basic Idea

- Maintain a sparse address space of roughly twice the total size of server capacities for easily allocation and deallocation
- For each server, continually hash its ID until you reach a contiguous space of the size equal to the capacity the server s_i with capacity c_i, and then allocate that space to the server (For example, $h(s_i)$ to $h(s_i) + c_i$, $h(h(s_i))$ to $h(h(s_i)) + c_i$, $h(h(h(s_i)))$ to $h(h(h(s_i))) + c_i$, etc.)
- To hash some video request r, keep hashing until you reach a valid address of a server, and assign. (For example, $h(r)$, $h(h(r))$, $h(h(h(r)))$, etc.)

3.2 Other Considerations

3.2.1 Handling New Servers

To handle the situation where you are adding another server, you can simply allocate a contiguous space of size equal to its capacity just like above. Then if any video is routed to this new server, add that video to its cache, and find where the video was previously cached by repeatedly applying the hash function until you land in another server's address range (assuming there was a previous request for v, which can be checked with a Bloom filter).

You may run into an issue if you cannot find a sufficiently large contiguous space for the new server, but we can minimize the likelihood of this issue occurring simply by allocating a large enough initial address space (but making the initial address space too large could mean that you have to try many times to hash a video to a machine). You could attempt to split up the server address space in order to avoid this issue, but this could lead to fragmentation, a common issue in memory management.

3.2.2 Handling Crashed Servers

To handle the situation where a server is crashed, mark its address space as now free, and for each video in its cache, repeatedly apply the hash function in the same fashion as above until you land in the range of an active server, and add the video to that other server's cache.

3.2.3 Cycle Issue

An issue may arise due to the way in which the hash function is continuously applied to a video request r. You may encounter a cycle, where you continue not to find a server, and then you loop back to the original hash $h(r)$ (For example, $h(r) = h(h(h(h(r))))$ and $h(r)$, $h(h(r))$, $h(h(h(r)))$, all do not hit valid server addresses). In this case, you may run into an infinite loop, never able to find an valid server to hit.

The solution is to modify the hashing process. We can incorporate the length of the current path into the hashed value. That is, instead of $h(r)$, $h(h(r))$, then $h(h(h(r)))$, we will use a hash function that takes in two values, and we will get $h(r, 0)$, $h(h(r, 0), 1)$, $h(h(h(r, 0), 1), 2)$, etc. This will avoid this issue entirely, as the second value will always change.

3.3 Performance Analysis

Now let's consider the run-time of this algorithm, which is based on the expected number of locations we must check for a valid server address.

First, we must consider the probability of hitting a valid server address. If we assume that our hash function is universal and all servers are allocated randomly across the hash range, it is a simple conclusion to make that the probability is simply the total size of valid server addresses divided by the total hash range size. If, as suggested, we have a hash range that is 2x the total capacity of the servers, then we get that the expected number of tries until we hit a server address is just 2.

We can equivalently think about flipping a fair coin, where the probability of Heads is $1/2$, and thus, the expected number of flips until you get a Heads is 2.

This means that we have $O(1)$ run-time, as opposed to $O(logn)$ in consistent hashing.

4 Another Constraint: Popularity

We must also consider that some videos might become much more popular than others. Specifically, Yahoo! might need multiple front-end servers to handle one very popular video.

Initial thinking might suggest that a Bloom filter could handle this problem of set membership. However, over time, as Bloom filters cannot easily handle deletions, any video may become popular. How do we handle this issue?

Solution: Multiple Bloom filters (The Zebra System)

- Maintain a set/array of Bloom filters
- Every day, clear out the videos in the oldest Bloom filter
- Then, add the currently most popular videos and insert them into the latest Bloom filter
- Over time, videos that were popular but are no longer popular will eventually get discarded
- Then, you can check if a video is popular by checking for the video in the union of all of the Bloom filters.

5 Results

In the Yahoo! Video Platform, the Zebra system was used to select a cluster of servers to handle a given request (based on factors like popularity and physical locality), and the SPOCA algorithm was used to determine which server among the cluster would serve the request (potentially from its cache). This system decreased cache miss percentage from 9.7% to 0.4% over about two weeks, saving $350 million in unnecessary equipment costs alone over five years.

References

[1] Ashish Chawla, Benjamin Reed, Karl Juhnke, and Ghousuddin Syed. Semantics of caching with spoca: A stateless, proportional, optimally-consistent addressing algorithm, 01 2011.