Consistent Hashing Final Report

CS5150/CS6150 Advanced Algorithms Semester Project

Jeremy Whipple

Michael Zhang

Simon Redman

Introduction to the paper/topic

We are interested in taking a different approach to solving the core problem of distributed load-balancing discussed in *Consistent Hashing and Random Trees* by Karger et. al [1]. On the internet, there are often intermediate cache nodes. These cache nodes store usually static content at a location that is logically or physically closer to the end-user, thus reducing delays. These caches also reduce the load on the home server which might not otherwise be able to handle high demand (aka, hot spots/pages/keys). However, these caches might themselves be subject to unusually high load and also be overloaded.

The problem the paper aims to solve is to distribute content across cache nodes such that none is doing more work than another. This essentially means the clients need a smarter way of selecting the proper cache for their request. To this end, the paper proposes a consistent hash function which performs well in cases where the client nodes might not have a complete view of the network, and which supports new cache nodes being added to a particular view with the minimum perturbations to the hash outcome. The paper also defines how a hash function can be "consistent", and what implications that has. Specifically, the paper describes how a tiered cache hierarchy can be constructed through a consistent hashing scheme with random trees.

Formal description of the algorithmic problem

When a server that hosts a popular web page is swamped with requests, a common technique is to add a layer of caching in front of the server. However, if the rate of requests is high enough, then this layer of caching itself can become swamped. A naive solution would be to add a layer of caches in front of the swamped caching layer. By applying this approach recursively, we get a hierarchically layered set of caches similar to a tree. However, how do we decide which cache to look up, and how is this possible when different users/browsers view of the internet and the set of caches is different?

The solution in the paper uses two tools, a randomly-constructed path through a tree of caches to try to find a node which has the desired cached data, and a consistent hash function to ensure that: 1) clients are able to identify the content they are looking for even if they do not have a complete view of the network, 2) with high confidence, swamping of nodes does not occur, and 3) nodes are able to enter and leave the cluster incrementally with minimal key movement. Formally, a hash function is consistent if it meets properties of balance, monotonicity, spread, and load. These methods seem to do a good job of balancing load and handling client requests.

Algorithm provided in the paper

More concretely, the consistent hashing algorithm proposed in the paper divides a unit interval up into C pieces, where each cache is responsible for a portion of the keyspace. This can be done (for example) by hashing the IP address of each cache. When a key-value pair is inserted, the key is hashed and stored at the responsible cache. The responsible cache is the cache whose IP address is closest to the hash value of the key. When a key-value pair is fetched, we hash the key and now know exactly which cache the key should have been stored at. When a node enters or leaves the cluster, only the keys local to the keyspace of the node and its neighbors can potentially be reshuffled. The paper presents proofs on why this scheme is consistent.

Random cache trees are used to coalesce requests. The root of the d-ary tree is the server that serves requests, and the internal and leaf nodes are caches in the cache hierarchy. When a browser wants a page, it picks a random leaf to root path in this tree and asks the leaf for the page. When a cache receives a request for a page, it will satisfy the request if it currently stores the page, or it will escalate the request up to its parent if it doesn't store the page. This request can eventually percolate up to the root server if no cache along the path has cached the page. A cache will cache the page if it receives enough requests for the page. Note that the random tree is constructed by hashing the key of the page we are looking for. Therefore, each page has a unique random cache tree consisting of the caches the browser knows about.

Taken together, random trees and consistent hashing solve our problem with relieving hot spots on the internet. A tree hierarchy of caches is formed for each page, and the mapping from each tree node to an actual physical machine containing a cache is determined using consistent hashing. This allows different browsers with inconsistent views of the available caches to use random trees effectively in a dynamic environment like the web.

Other baseline algorithms

A naive approach to load balancing between caches is to use traditional modular hashing. This is a bad idea because of how dynamic the set of available caches are in a distributed system like the internet. Every time the set of caches changes, the mod factor of the modular hash function would have to change in the same way. This will cause an almost complete reshuffling of the assigned keys since each key would very likely map to a new cache. As a result, the network contention would be huge as each cache frantically tries to send its keys to another cache.

A naive approach to hierarchically caching would be to use a single deterministic tree for all the pages. This isn't a great solution since then the caches closer to the root server are more likely to be swamped. By creating a unique random cache tree for each page, the requests and caching responsibilities can then be distributed more evenly across all the available caches.

Note that our approach differs from *Consistent Hashing with Bounded Loads*, where the authors propose skipping caches in the ring until one is found that has enough capacity/bandwidth [4]. Another approach that has been taken by *Dynamo: Amazon's Highly Available Key-Value Store* is discretizing the consistent hashing ring [5]. These are both other alternatives that exist in the literature.

Our Implementation

Using Consistent Hashing and Redirection, we have implemented a load-balancing protocol which has interesting parallels to cuckoo hashing [2]. We have implemented Consistent Hashing in the manner described in the original paper [1]. Every cache is assigned a token uniformly at random. When a client wants to request a file, it applies a sha256 hash function to the file, finds the cache who's token defines the range into which the file should fall, then makes the request.

Every file has two hash functions which are computed by appending a salt to the file's contents. Since sha256 should be uniformly distributed even for only salted inputs, this effectively gives us two totally different ways to hash a particular file. This means, in our implementation, a file should reside on exactly two caches. The client makes a request to both caches asking whether it actually has the file and its current load. The client then downloads the file from whichever cache has the file and had the least load. This is illustrated in Figure 0.

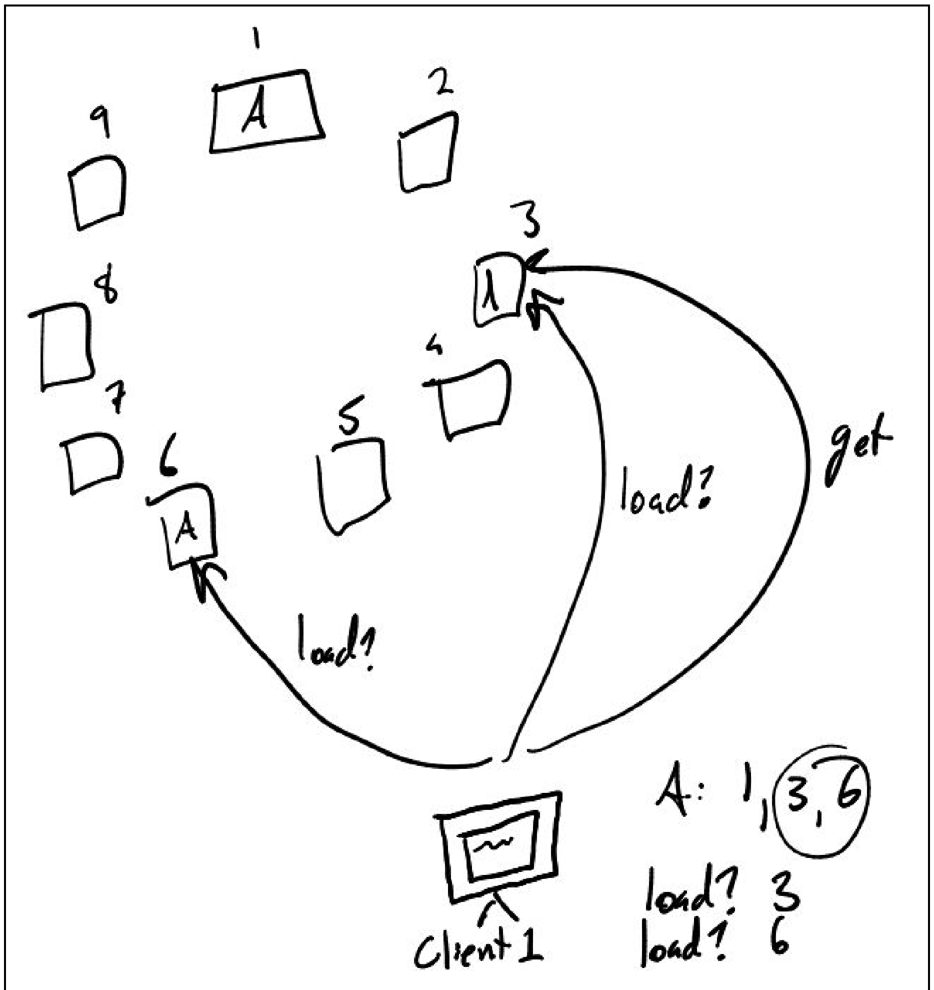

Figure 0: Ask-First Randomized Load Balancing

If a client has an inconsistent view, it may make requests to the wrong cache. For our simulation, we assume that if the client misses on both hash functions it goes directly to the main server hosting the actual file. If a particular cache gets many requests for the same file, it might eventually decide to download and cache the file for future requests. Specifically, this threshold is chosen using the *max_misses* flag of our simulator.

An alternative way to implement this would be that the client could then select a random node from the possible caches containing the file to query. However, as shown in class, random selection has a decent probability of delivering a significantly higher load to some nodes. We were inspired by the solution presented in class which suggests randomly selecting two sources then delivering the request to the one which is least loaded. This is akin to the thesis of *The Power of Two Choices in Randomized Load Balancing* [3]. One pitfall of this solution is that the load-querying request itself adds load. However, we assume is that the content is large, thus a small query is relatively insignificant.

We have also implemented a rebalancing mechanism for use when a particular cache is overloaded. Consider one node which is hosting multiple pieces of content. The node can reduce its own load by offloading a content item to another node. Since we are assuming a client-server model, where the server has no way to update the client, the client will still make its request for a piece of content to the original cache. The cache then simply replies to the client with a redirection notice. As with the other solution, this has the problem that the original request and the redirect request have a cost, but as before we assume this cost is minimal in the overall picture. Such costs are taken into account in the simulator and can be changed using the *file_size* flag.

Taken together, we have combined these two approaches into a protocol that has similar properties to cuckoo hashing. Two hash functions are used to map keys into two caches in the consistent hashing interval. When a client gets a key, it asks both caches in parallel. Whenever a cache detects that it's overloaded, it can move some of its keys to the key's alternative caches using the other hash function. This is similar to a cuckoo bird pushing eggs out of the nest and results in a fine-grained load balancing scheme that makes efficient use of resources. This approach is a similar and simpler approach compared to random cache hierarchy trees and retains nice properties with respect to distributing load evenly across caches. A potential pitfall is the extra client get and put communication. However, the benefits to reduced load make this a good trade-off. The end result is clients will hopefully have a faster and/or more available way of fetching their keys.

Experiment Setup

To evaluate this approach, we have implemented both consistent hashing and consistent cuckoo hashing and test them on a synthetic workload dataset. Due to most read and write workloads following a Zipfian or Pareto distribution, we have randomly generated a workload that puts and gets keys following a Zipfian/Pareto probability distribution. This would challenge any sort of load balancing scheme since the popularity of each key is not the same. By measuring and plotting over time the max cache load to min cache load ratio and network overhead of each approach, we hypothesize that consistent cuckoo hashing will outperform vanilla consistent hashing. The results are also compared to a uniform distribution of key popularities.

For simplicity, our experiment assumes that every request/response can be completed in one “cycle”, that all files contribute the same load, and that all the caches have the same ability to handle requests.

When cuckoo hashing is added on top of consistent hashing, the client queries two caches using two different hash functions. Depending on which cache stores the actual file, that cache will respond. Therefore, file size matters a lot since we are effectively doubling the queries we perform. However, if the file sizes are large, then this can be beneficial in distributing load since only one cache needs to respond, and the cache that is more loaded can simply drop the request. The additional network traffic/bandwidth is reported in the load statistic in our graphs.

Finally, our experiments take into consideration “inconsistent views”. Realistically in a dynamic environment like the internet, all clients probably have differing views of which caches are available. Consistent hashing addresses this directly due it properties relating to spread and monotonicity. The simulator we have created lets the user choose the number of clients, number of caches, and importantly for “inconsistent views”, the number of visible caches for each client.

Results

Figure 1 and Figure 2 show some example results from our experiments. In Figure 1 we see that over 100 iterations/simulation timesteps, our load balancing scheme is able to maintain a fairly balanced load between the most- and least- loaded cache. Notice that, with Cukoo enabled, the maximum difference in any cycle between most- and least- loaded cache is about a factor of 4.25, while with Cukoo disabled, the balance is between about a factor of 15 and a factor of 100. Most of our results look like Figure 1. However, in some cases, using the same parameters to the simulation, the balance is terrible, as shown in Figure 2. Notice that our Cukoo methods are able to maintain the balance a little bit better, but not a lot. We have not directly investigated what circumstances lead to this kind of behavior. Since it is infrequent, it is likely a result of an unlucky situation where one cache gets randomly assigned to a very large portion of the keyspace. The number of caches in these experiments (for simulator runtime reasons) is 10. We hypothesize that as the number of caches grows, the chance of this unlucky assignment of keyspace will diminish.

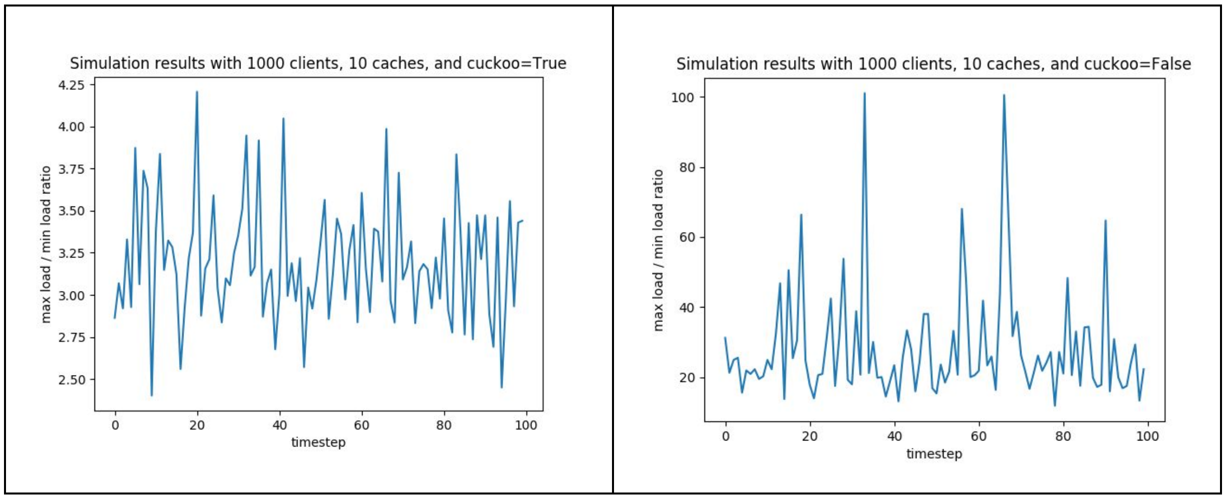

Figure 1: Results from a simulation showing a successful balancing operation. File size = 400KB and each cache can services 1Gb/cycle

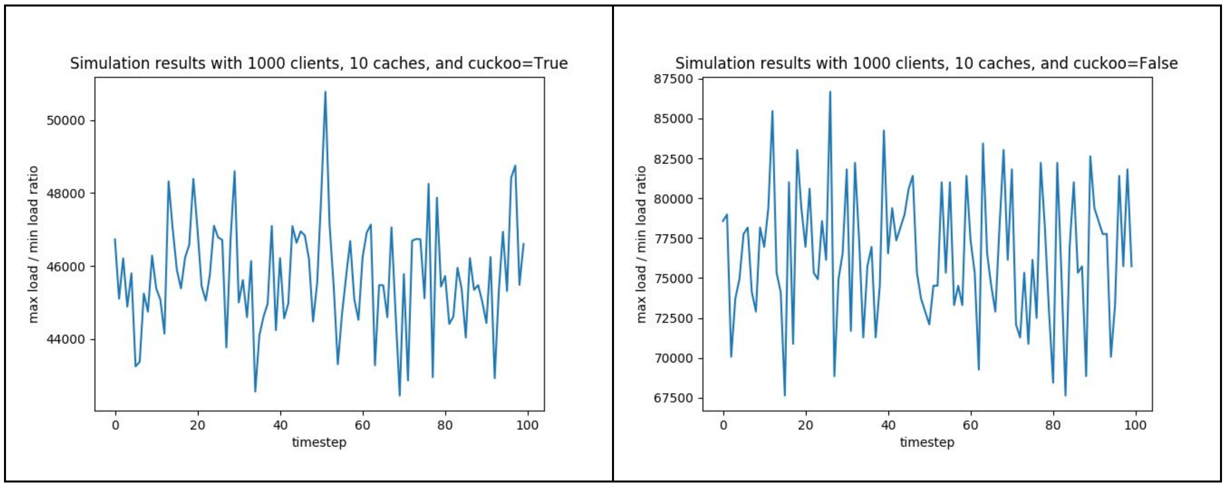

Figure 2: Results from a simulation showing unsuccessful balancing. File size = 400KB and each cache can services 1Gb/cycle

Figure 3 shows what happens when the distribution of request follows a Pareto/Zipfian distribution. In short, requests for files are skewed a lot depending on the popularity of the file, and each subsequent file is about twice as popular as the next. As you can see from the simulation results, this insanely skewed distribution caused a small increase to the load ratio when cuckoo hashing is used. Meanwhile, the skewed distribution caused a huge increase to the load ration when cuckoo hashing isn't used. This shows how important it is to choose a cache depending on how much load is on it. Even though some caches are responsible for files that are way more popular, the resulting load ratio is minimized since it is likely the loaded cache does not need to do anything except host that one file. It can effectively drop all other requests and defer them to the second responsible cache. Compared to the uniform distribution case, consistent cuckoo hashing has shown its benefits at mitigating hot spots.

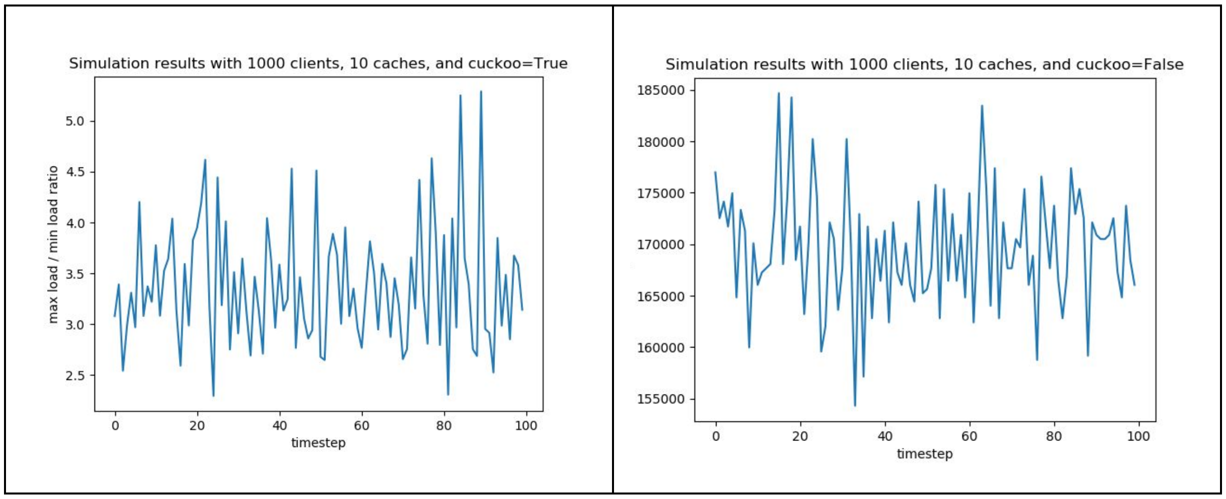

Figure 3: Results from a simulation with a Pareto distribution of the requested files. File size = 400KB and each cache can services 1Gb/cycle

Figure 4 shows what happens when the clients see an inconsistent view of the caches. In the experiments shown below, each client only knows of a random subset of the caches. Normally with modular hashing, this is undesirable since each key regardless of hash value gets uniformly distributed amongst the known caches. However, with consistent hashing and its spread and monotonicity properties, each key is associated with the closest cache in the consistent hashing ring. Therefore, even though the view of what caches are available are inconsistent between clients, the key is not spread between too many caches. Even under inconsistent views, the performance is on par with previous experiments. Note however that the relative magnitudes of load ratio are important, whereas direct numerical comparison is not. This is because the initial randomness in assigning caches to portions of the keyspace and keys to popularities adds a lot of noise in the experimental measurements.

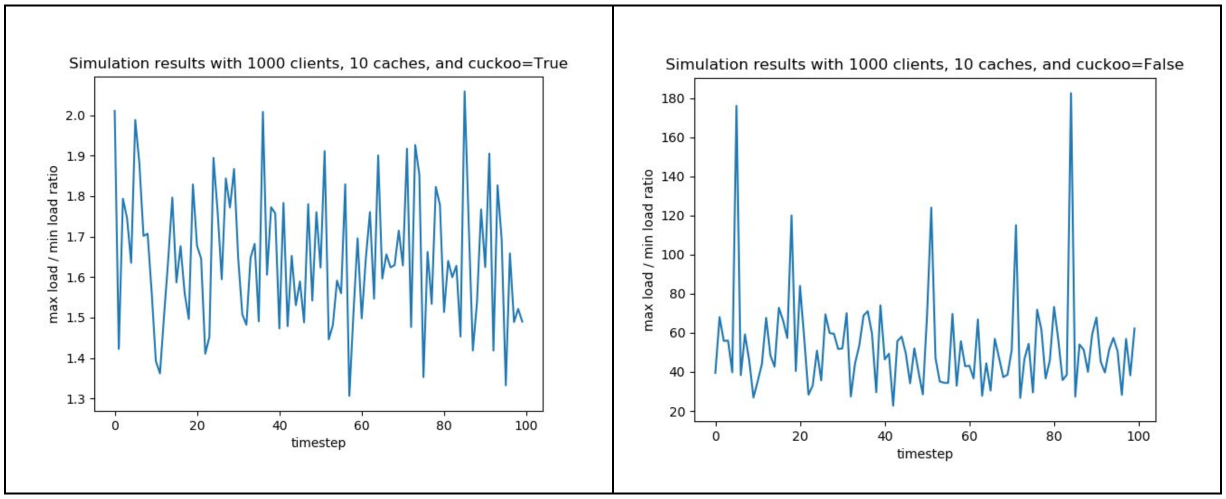

Figure 4: Results from simulation with a Pareto distribution of the requested files and an inconsistent view of the world. File size = 400KB and each cache can services 1Gb/cycle

Conclusion

As we can see from the results, our load balancing mechanisms do a good job maintaining fairly even load between the most- and least- loaded cache nodes. There are still some cases where the load is incredibly unbalanced, but this is why the original paper decided more complicated methods like random trees were needed. For being significantly simplified, our methods perform admirably.

However, this also depends on our network assumptions being valid, which may not always be the case. Also, the noise introduced by the randomness in our experimental setup made strict comparison somewhat difficult, and the variance between runs was sometimes considerable. To obtain better measurements, it would have been better to run the experiments longer and in an actual environment like a mocked version of the internet instead of the relatively basic simulator we created. However, this would have required compute resources and engineering resources beyond the scope of a course project.

To summarize, we blended consistent hashing with cuckoo hashing and simulated how such a protocol would behave when subjected to a uniform distribution of keys, a popularity distribution of keys that varies wildly, and in an environment where clients have inconsistent views of which caches are participating. We found that indeed cuckoo hashing and consistent hashing did good jobs, under our assumptions, at minimizing load ratios and handling inconsistent views. We have learned a lot in this course project. The code can be found at:

https://github.com/michael13162/ConsistentHash. Instructions for running our simulator and descriptions of the flags are in the README.

[1] Karger, David, et al. "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web." Proceedings of the twenty-ninth annual ACM symposium on Theory of computing. ACM, 1997.

[2] Pagh, Rasmus, and Flemming Friche Rodler. "Cuckoo Hashing." Journal of Algorithms 51.2 (2004): 122-144.

[3] Mitzenmacher, Michael. "The Power of Two Choices in Randomized Load Balancing." IEEE Transactions on Parallel and Distributed Systems 12.10 (2001): 1094-1104.

[4] Mirrokni, Vahab, Mikkel Thorup, and Morteza Zadimoghaddam. "Consistent Hashing with Bounded Loads." Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms. Society for Industrial and Applied Mathematics, 2018.

[5] DeCandia, Giuseppe, et al. "Dynamo: Amazon's Highly Available Key-Value Store." ACM SIGOPS operating systems review. Vol. 41. No. 6. ACM, 2007.

NOT INCLUDED

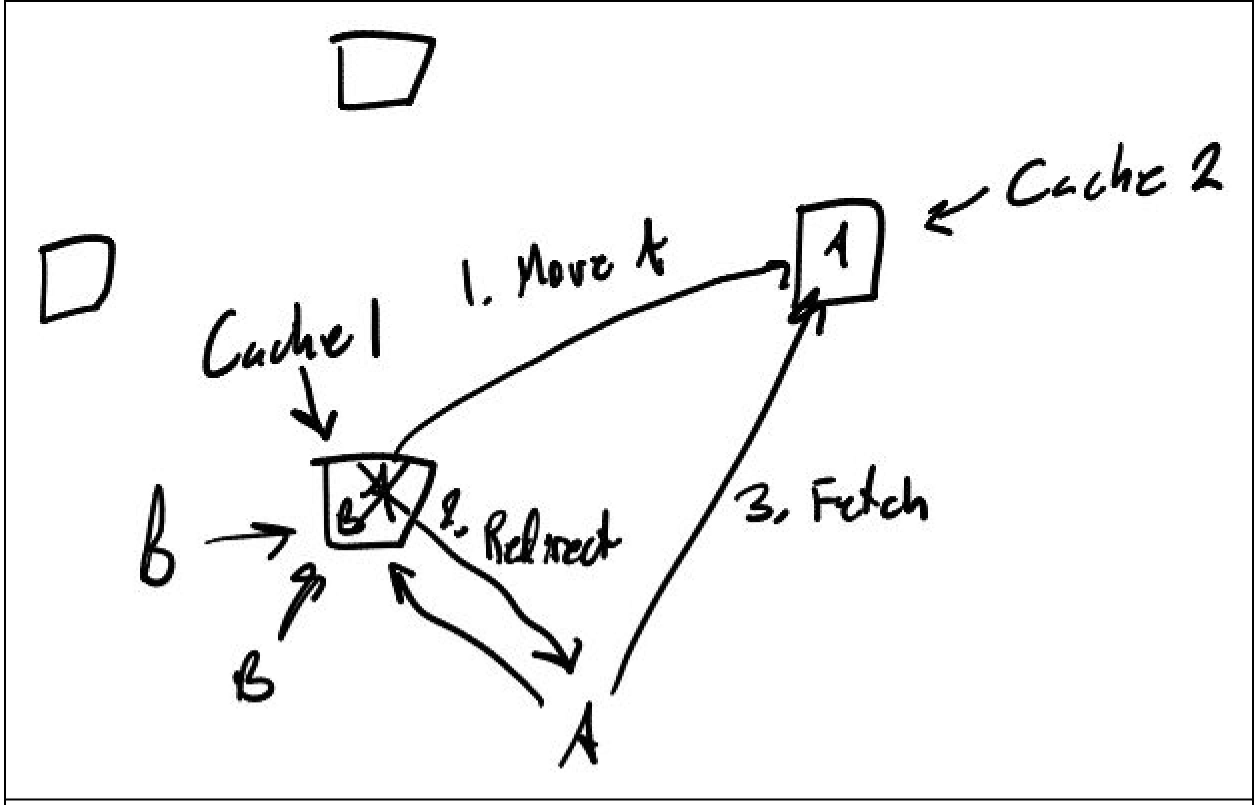

Figure 4: Redistribute and Redirect Load Balancing