Static and Dynamic Load Balancing Techniques and Implementations

Johan Vikström - jovi@kth.se
Georgios Tagkoulis - giorgost@kth.se

April 2020

1 Introduction

Load balancing is an integral part of any distributed system that is mostly relevant during operations. It is not only integral to being able to increase capacity but also a requirement to improve resiliency. One of the big problems distributed systems tend to have are hot spots, servers that get a much higher load than the mean load. Hot spots in a system generally cause bad tail latency [1]. So the main goal with load balancing is to remove those, or rather, the main goal with load balancing is to reduce the load imbalance between different servers in a system [2]. In a data center setting this over-utilization and corresponding under-utilization of parts of the system can not only lead to decreased response times in both the tail and in general. But it can also lead to increased costs for the cooling systems and in increases of CO_2 emissions [2][3]. So poor load balancing does not only cause systems to run slower, it can cause systems to violate their SLOs, increase costs and increase CO_2 emissions.

When talking about load balancing the first thing one thinks about is normally about load balancing the computational load, for example load balancing http request/responses. But systems also need to balance storage resources (i.e. sharding) to optimize the medium's size, IOPS and resiliency [4].

Load balancing is generally classified into one of two groups depending or not if they have knowledge of the current state of the system while executing. Either static or dynamic [2]. Static load balancing does not depend on and has no information about the current state of the system in contrast to dynamic load balancing.

For basic request/response workloads random balancing takes you a long way to get really good load balancing. However for certain tasks that require a lot of preloaded data, for example think tasks requiring machine learning models, something that is more application aware is required. If one would try to do random load balancing for this the request latency would be dominated in pulling the model. One example of such a system is Slicer which Google created as a general purpose auto-sharding service. The median Slicer managed workload uses 63% less resources than if they would use another form of static load balancing [4]. So thought out load balancing can lead to big cost savings and decreased response times.

In this essay we will present a few different approaches to load balancing and give examples of where they are used.

2 Static load balancing

The goal of every load balancing technique is to distribute the workload evenly to all the available nodes/workstation, either it is in cloud computing or databases and web cache management.

Static load balancing depends heavily on the prior information about the system [3], and requires a master node, centralized router, to distribute the workload to the available nodes. This type of load balancing is best suited to homogeneous systems [2]. Since there are no dynamic statistics about the current state of the system (workload), the process assignment is decided at the beginning of the execution and cannot be changed later on. This means the static load balancing algorithms are non-preemptive [2]. There are advantages and disadvantages in this approach. The lack of any communication between the nodes aims to provide less overhead in each node resulting to reduction of communication delays and minimization of the process execution time [2]. Two of the most notable static load balancing algorithms are the Round-Robin and consistent hashing algorithms.

2.1 Round-Robin

The Round-Robin algorithm is one of the simplest algorithms for distributing the load to a set of nodes. The idea is the master node to assign each load to the list of nodes N_1-N_n based on a rotating order, so eventually the jobs are evenly distributed to the available slave-nodes [5]. Thus, the first load goes to N_1 node, the second to N_2 and so on. Only after a load is assigned to the N_n node, the master node will assign again to the N_1. While simple and efficient in many cases, its main drawback is that it does not take into consideration the execution time of the incoming load, meaning that the system could end up in some nodes being "swamped". Therefore, it is more suitable for requests that are similar in nature, like the http requests in web server application [5]. With prior knowledge of the execution time of the incoming tasks, Round-Robin could be modified with weights so the heavy requests could be directed to the most powerful nodes.

2.2 Consistent Hashing

The consistent hashing algorithm was introduced in 1997 [6], in an effort to provide a better implementation of distributed caching for web servers. In the case of web servers, the resources the web server serves are evenly distributed to a number of caches. This means clients can fetch the resources without overflowing the server with requests.. Both the server and the clients have the same hash function, so the correct resources are served each time. Consistent hashing does not use a classical hash function, like the linear congruential function $f(x) = a * x + b$ (mod p) where p is the number of slave-nodes (caches). The reason is that this would cause a severe deterioration of the system every time a node (cache) is added or removed from the system. The positions for all cached data would need to be re-calculated and sent to the new positions and in the mean-time all cached data would be useless since the clients would be searching for them in different positions [6].

Consistent hashing uses the notion of "views" to tackle this problem. "View" is defined as a set of caches that a particular client knows about, so each machine is constantly "aware of a constant fraction of the currently operating caches" [6] and every client uses a consistent hash function which maps the resources to one of the caches in their view. Two important properties of consistent hashing is "smoothness" and "spread". The former property refers to the minimum expected resources that must be moved each time a new node (cache) is added or removed, while the latter one refers to the

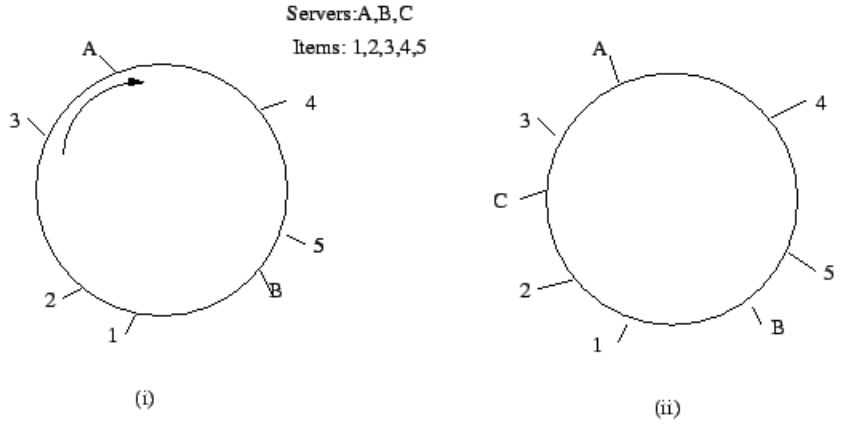

Figure 1: Example of consistent hashing when projected in a unit circle, where a number of URLs are mapped to the available caches. Figure adapted from [7] with the following explanatory caption: *(i) Both URLs and caches are mapped to points on a circle using a standard hash function. A URL is assigned to the closest cache going clockwise around the circle. Items 1, 2, and 3 are mapped to cache A. Items 4, and 5 are mapped to cache B. (ii) When a new cache is added the only URLs that are reassigned are those closest to the new cache going clockwise around the circle. In this case when we add the new cache only items 1 and 2 move to the new cache C. Items do not move between previously existing caches.*

small number of different caches that a resource exist from the perspective of the client "View" [6].

In 1999, Karger et al [7], provided an example of the consistent hashing algorithm with the unit circle as presented in Figure 1.

One thing to be noted about consistent hashing is that it's not better than just doing a random assignment of clients to server [8]. So there can be a lot severe load imbalance in a system using consistent hashing [9].

3 Dynamic Load Balancing

Unlike the static load balancing, in dynamic load balancing decisions about the distribution of the workload are influenced by the current state of the system. Many attributes are taken into consideration. Depending on the system these could include CPU power, CPU usage, memory, network bandwidth and so on. Loads can also be assigned to a particular node and later re-assigned to another node depending on the run-time information that is collected while the tasks are executed [2]. Despite the overhead that is added to the system due to the run-time communication among the nodes, dynamic load balancing techniques offers a number of advantages over static ones. First, in the case that a node is down, the system is not halted despite some deterioration in its performance. Furthermore, since it is possible to move a task from an over-utilized node to an under-utilized one, it is said that the dynamic load balancing technique is preemptive. At last, algorithms in this category are more versatile than the static ones, offering better outcomes in heterogeneous and dynamic conditions [2].

As dynamic load balancing may move items between nodes *key churn* is another important metric to optimize against. Key churn is a measure for how often and how many keys are moved during operations. While a key is being moved the data for that key might not be served depending on implementation. More importantly a high key churn will consume a lot of bandwidth as items are moved [4] [8].

3.1 Load Aware Consistent Hashing

One example of dynamic load balancing is a load aware consistent hashing algorithm. As in the normal consistent hashing algorithm we have a set of servers, clients and a hash function. The "Consistent Hashing with Bounded Loads" algorithm layered on top guarantees that none of the servers receive c times the average load (i.e. max load is cm/n where m is the number of requests, n number of servers) [8]. There are two main changes in this algorithm when compared to "normal" consistent hashing.

- Each server is given a capacity.
- If a node is full it will forward to the next server in the ring.

The algorithm to insert into the system becomes: *If a request is hashed to a server that is full, forward to next server in the ring. Else insert into the server.* Or in other words: it's exactly the same as one would do *linear probing* in hash maps (if server capacity is set to one). Consequently: when we search for an item in the system we begin by addressing the server that it's hashed to. If the item is in that server, we are done. However if it's not there and the server is full, we look at the next server in the ring and keep on doing this recursively. If we get to a server that is non-full and does not contain the item we are looking for - the item is not in the system and the find can terminate.

This means that when deleting we can't simply remove an item. This could cause the find algorithm to fail if an insert has propagated any item. So if a node is full and an item is deleted the algorithm might need to make sure it's still full. We say that the node has a "hole" where the item used to be. There is a very simple recursive algorithm for solving this: *If the server propagated an item i, grab i from the next server and fill the hole, continue doing this recursively with the next server.*

When a server is added to the system the "back propagation" algorithm in the last paragraph for deleting is used. Simply mark the server to have as many holes as it has capacity and start filling holes in the entire system for as long as possible.

One thing to be noted about this is that capacities can be set independently of each other, i.e. one server can have a capacity of 1 and another 10 if that is required for some reason.

This algorithm is used by Vimeo to load balance their video servers, previously they had used consistent hashing but they were getting a lot of load imbalance even after trying a lot of different variants of consistent hashing that was supposed to solve this. However after moving to load aware consistent hashing the problem of load imbalance was solved [9]. It was also previously used by Google's internal Pub/Sub, however now it instead uses Slicer [8].

3.2 Slicer

In this section we will give an example of a dynamic distributed load balancing system to hopefully make it clear how they can be implemented and just how much more complex than static load balancing it can be.

The system we will describe is Slicer. Slicer is a Google general purpose auto-sharding service used by internal Google applications. It's a very versatile system being used for a lot of things from per-user caches to topic based pub-sub to web-crawl manager (saving site metadata) to serving fonts (which currently servers about a million fonts a second). When the Slicer paper was written it served 4-7 million requests per second. The systems load balanced with Slicer use 63% less resources in median than they would if using some static load balancing algorithm [4].

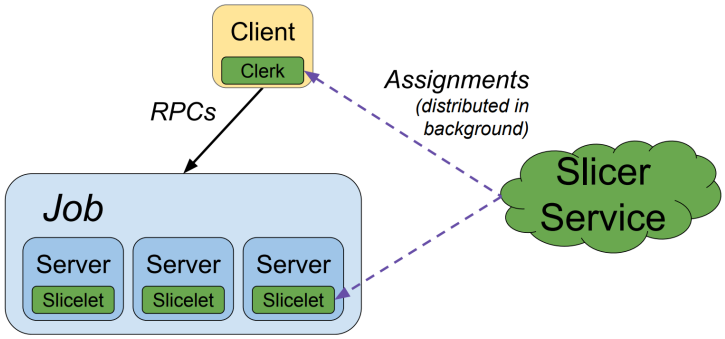

Figure 2: System overview: The Slicer service sends slice assignments to the client and server slicelets.

Slicer is maintained as a separate service which allowed them to decouple their release schedule from the projects that depended on them. The API they provide to services that want to utilize Slicer load balancing is on the order of 5 functions and only 1 function is required to integrate into Slicer, namely mapping a key to a server address. But this function has been integrated into their RPC system meaning for most cases this mapping is handled transparently [4]. This means that a lot of the complexity is not surfaced to application developers which makes integrating Slicer relatively simple.

3.2.1 Architecture

Slicer is an entire distributed system by itself with a few different parts. But before explaining the architecture we need to define two things: a *task* is a single application process, a *job* meanwhile is a collection of *tasks* being load balanced in a data center. Slicer contains of three primary components: the *slicer service*, the *clerk* which is library linked into clients and finally the *slicelet* which is a library linked into the servers. The slicer service generates slice assignments and and notifies the slices and clerks of them. Applications only interact with slicer using the clerk and slicelet libraries. In Figure 2 from the Slicer paper there is an overview of how these components react.

It's also quite interesting for us to look at the actual *Slicer Service*. The component generating slice assignments is the *Assigner*, it takes different signals from the system as input, generates and sends the assignments to the store and distributor. One important part is that there may be multiple Assigners in the same system. The way we guarantee that assignments do not diverge is in the Store. Before generating an assignment the Assigner first fetches the current assignment and updates the current one. If there's been an update while it was computing it's own updated assignment it retries with the new assignment. As slicer does fine-grained load balancing over billions of keys sending assignments to slicelets and clerks becomes a bottleneck. The distributors are there to reduce the network load. Essentially they become a proxy for a set of clerks/slicelets. This way the distributor can be placed close to the clerks/slicelets that are subscribed to it which will reduce the data center "bisection" network usage. As instead of sending assignments over the entire data center the data will only be local to the rack or set of racks. This is all illustrated in 3 from the Slicer paper.

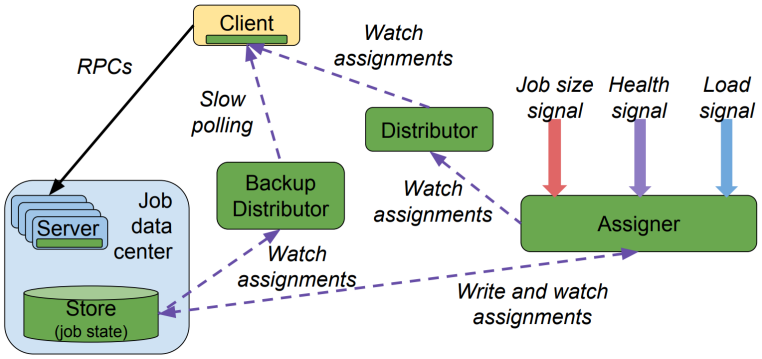

Figure 3: Slicer service architecture. The *assigner* collects signals and uses them to make slice assignments. The assignments are then sent to *clerks*/*slicelets* via *distributors*.

3.2.2 Assignment generation

The goal Slicer optimizes against is to reduce load imbalance, maintain a high availability and minimize churn. This is achieved by load balancing over application defined keys. Each one of these keys are hashed to 63-bit *slice keys* and then Slicer load balances ranges of these slice keys. This range based load balancing makes the Slicer work load independent of the number of keys - meaning the application can have a billion keys or 10 keys, Slicer is still able to load balance just fine. The hashing of keys also reduces the probability of local hot spots as the resulting slice keys will be uniformly distributed over the entire 63-bit range. In addition, Slicer can assign each range to multiple tasks (i.e. servers) if a certain range gets has a high load. This might be useful for things like machine translation where more than a single task is required to serve translations to/from a specific language.

Initially the key space is evenly distributed amongst the tasks. During operations Slicer monitors key load using either internal Google infrastructure or application reported metrics. When slicer regenerates assignments it does two primary things to slices: merging and splitting. Finally it also moves the resulting slices in a way that will reduce load imbalance while minimizing key churn. The Slicer authors call this algorithm a "weighted move", it goes as follows:

1. Reassign ranges that are no longer part of a job (because of task failures for example)
2. Modify key redundancy in case configuration has changed
3. *Merge* adjacent cold key ranges into a single range
4. Pick a sequence of slice moves with as high a possible weight. Where the weight for a move corresponds to reduction in load imbalance divided by *key churn*.
5. Finally split slices without changing the task they are assigned to (These split slices can then be used the next time the assignment is done).

Those are the important parts of Slicer for our purposes has been covered, however the paper contains a lot more content. For example how the Slicer API looks like, how Slicer handles fault tolerance and how Slicer supports consistent assignments. It also contains a fairly extensive evaluation section from different production examples.

4 Conclusions

Load balancing is an extremely important aspect of distributed systems that directly affects the performance, complexity and maintenance costs of the system. Depending on the nature of the system, different approaches and techniques have been implemented. In the case of simple http request/response workloads static load balancing algorithms, like Round-Robin that was presented in a previous chapter, have shown that perform very well. On the other hand, for more advanced large scale workloads dynamic load balancing algorithms is a necessity for a well-functioning system. Well-designed load balancing systems, both in terms of the implemented algorithm and the API that is provided to application developers, is also clearly beneficial. Since one of the goals of DevOps is to deliver application quicker with fewer failures, it becomes apparent that load balancing plays an important role to that direction.

One example of success with well designed load balancing is Vimeo. They first tried to apply a lot of different band-aids into their consistent hashing load balancing without getting any better results. The desired results only came after completely redesigning their load balancing and moving to the load aware consistent hashing algorithm.

An example of a successful general purpose load balancing system is Google's Slicer. It has been a success with a large decrease of resource consumption. It's been applied to a lot of different services within Google, and the reason is clearly its very simple API that makes it easy to integrate because a simple API is one of the most important parts of a load balancing system. In general, a perfect load balancing system with a massive, incomprehensible API is probably worse than one that's a bit imperfect with a simple API - as no one will understand the one with the incomprehensible API.

References

[1] Yu-Ju Hong and Mithuna Thottethodi. “Understanding and mitigating the impact of load imbalance in the memory caching tier”. In: *Proceedings of the 4th annual Symposium on Cloud Computing*. 2013, pp. 1–17.

[2] Jitendra Bhatia and Malaram Kumhar. “Perspective Study on Load Balancing Paradigms in Cloud Computing”. In: (Mar. 2015).

[3] Sambit Kumar Mishra, Bibhudatta Sahoo, and Priti Paramita Parida. “Load balancing in cloud computing: A big picture”. In: *Journal of King Saud University - Computer and Information Sciences* 32.2 (2020), pp. 149–158. ISSN: 1319-1578. DOI: https://doi.org/10.1016/j.jksuci.2018.01.003. URL: http://www.sciencedirect.com/science/article/pii/S1319157817303361.

[4] Atul Adya, Daniel Myers, Jon Howell, et al. “Slicer: Auto-Sharding for Datacenter Applications”. en. In: (), p. 17.

[5] Zahra Mohammed Elngomi and Khalid Khanfar. “A Comparative Study of Load Balancing Algorithms: A Review Paper”. en. In: (2016), p. 11.

[6] David Karger, Eric Lehman, Tom Leighton, et al. “Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web”. In: *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. STOC ’97. El Paso, Texas, USA: Association for Computing Machinery, May 1997, pp. 654–663. ISBN: 978-0-89791-888-6. DOI: 10.1145/258533.258660. URL: https://doi.org/10.1145/258533.258660 (visited on 04/20/2020).

[7] David Karger, Alex Sherman, Andy Berkheimer, et al. *Web caching with consistent hashing*. May 1999. URL: https://doi.org/10.1016/S1389-1286(99)00055-9 (visited on 04/20/2020).

[8] Vahab Mirrokni, Mikkel Thorup, and Morteza Zadimoghaddam. “Consistent hashing with bounded loads”. In: *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM. 2018, pp. 587–604.

[9] ardoland, Vimeo Engineering Blog. *Improving load balancing with a new consistent-hashing algorithm*. 2016. URL: https://medium.com/vimeo-engineering-blog/improving-load-balancing-with-a-new-consistent-hashing-algorithm-9f1bd75709ed.