General Sharding Service

Rubens Farias, Harrison Lundberg, and Yumin Zhang

December 2017

I. INTRODUCTION

In this project, the goal was to design a centralized load balancing algorithm that is generalizable enough to be used with any sharded service. We wanted to design a load balancing algorithm that both minimizes load imbalance but also minimizes churn on each partitioning update. The workload we designed for was a large key space with non-uniform accesses across all keys, and a variety of operations on individual keys that have different costs. We also assumed that the workload could change dynamically, so the partitioning scheme couldn't be statically determined beforehand. For this work, we implemented the load balancer as a central service that monitors the load of a sharded key-value cache. As you will see in our results, our algorithm correctly balances the load as expected. By designing our algorithm in a way to balance the load of multiple types of operations on a large key space, we believe that our load balancing algorithm can be used in a variety of contexts needing a content-aware load balancer.

II. RELATED WORKS AND MOTIVATION

When deciding how we wanted to manage partitioning, the first option considered was consistent hashing. However, as noted in the Dynamo paper, pure consistent hashing has limitations such as unequal key distribution. We then looked at the ways consistent hashing was extended for Dynamo, such as virtual nodes and fixed sized partitions. We also looked at the Slicer paper, which identifies a shortcoming of consistent hashing in that it gives less control over hot spots. Slicer uses a centralized Slicer Service that monitors the load on each node and makes load balancing adjustments with a weighted move algorithm. Additionally, Slicer uses variable sized partitions.

After considering implementing a variant of consistent hashing like Dynamo did, we decided to research and develop an algorithm that would function like Slicer's weighted move. Because we would be working with an algorithm like Slicer's weighted move, we also decided to handle all load balancing decisions in a centralized location.

We decided to use variable sized partitions as they allow us to better address hot spots by splitting a hot partition when necessary. The alternative would have been to use fixed-size partitions like Dynamo. However, the only way to deal with a hot spot would be to decrease the size of every partition and then remap the partitions to nodes to better balance the load.

One assumption we made about the sharded service using this load balancer is that for every request, a key can be extracted allowing the request to be mapped somewhere in the keyspace that we are partitioning. This mapping of request to key can be done explicitly by the client application or by a rule the service defines. For this project, the key for a request is defined by the application, as it is whatever key the operation is on in the cache. This is very similar to the assumption about services that Slicer makes.

III. ARCHITECTURE

A. KVCache

Each KVCache represents a single node in a distributed key-value cache, like a node in Memcached. A KVCache presents the client with a simple API for executing operations on a single key at a time. An operation is defined by its cost and the key it operates on. Each cache has a mapping of the key ranges it is responsible for and it services requests for these keys. Each cache will process the requests and record metrics on its overall utilization as well as the heat of each of the partitions assigned to it. A partition is simply a range of the key space. The heat of a partition is the sum of costs for all the operations on this key range during the last measurements period, which we set to be 1 second long. The KVCache is essentially a mock executor of client operations. The actual implementation and servicing requests is irrelevant as we are merely using the KVCaches to model request load, and not building a real functioning key-value cache service. Because each KVCache doesn't actually execute operations, it has to measure its load artificially by looking at the pending requests and their costs, and seeing how much of the current time interval must be used to service these requests, giving us the cache's utilization. It is possible for the cache to become overloaded in which case the requests queue up and utilization is at 100% for the current time slice. Every second, the cache sends metrics - utilization, total heat, and per-partition heat - to the central sharding service.

B. Sharding Service

Each KVCache periodically (every second) sends its load metrics to the central Sharding Service. Every 5 seconds, the service will aggregate each cache's most recent load metrics and begin to analyze them. The sharding service looks at all the caches' metrics, sees if there is a significant imbalance, and repartitions the key space in a way trying to reduce that imbalance. The algorithm used by the service

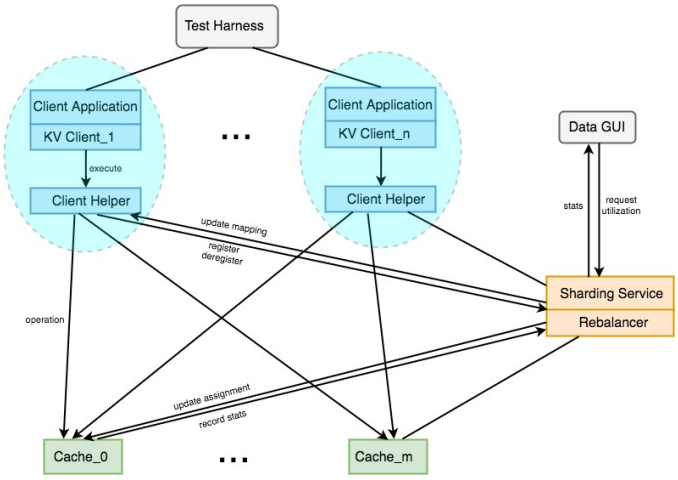

Fig. 1. System components

is described in detail later in the paper. When the service updates the partitioning scheme, it is said to update the assignment. An assignment can be thought of as the set of all key ranges (partitions) mapping to the one or more caches that serve their requests. When the service updates the partition assignment, it pushes the updated assignment to each cache node and client of the service.

C. KVClient

To allow an application to use our distributed key-value cache, we created KVClient, a class that any client application can use. This class presents a simple API to the application with an execute method where the client passes the key and the cost of the operation to be executed. The actual operation doesn't matter, just the key for routing and the operation's cost for load monitoring. At its instantiation, the KVClient registers with the Sharding Service to receive the initial partition assignment as well as to be pushed updated assignments in the future. When the application calls execute on an operation for a given key, the KVClient uses this assignment to route the operation to the correct cache node serving that key. The node is sent the key as well as the cost of the operation to be performed. Another implementation detail is the need for the ClientHelper. This is an actor that receives the updated mappings from the Sharding Service and is what communicates directly with the KVCaches. Any operation called on the KVClient library is first forwarded to the ClientHelper actor, who then communicates with the other components. Even though the KVClient and ClientHelper are represented as distinct rectangles in Fig. 1, they are logically just two threads running on the same machine.

D. Test Harness

To test our system, we created a Test Harness that starts up the Sharding Service, which in turn creates the cache nodes. We also created a basic Client Application that takes as parameters its request frequency and a Picker, which tells the application, for the current request, which key to access. We implemented a couple of different pickers, one being a Range Picker which uniformly selects one key from a specified range. We also added to Test Harness a command line API that allows the user to dynamically create different Client Applications to test how a given request load affects the partition assignment and the load (im)balance of the caches. The user can create and delete these applications to see how a changing request load forces the partition assignment to adapt.

E. GUI

To better observe the changing cache loads and the partition assignment, we created a GUI for our system. The GUI periodically pulls the most recent cache metrics from the Sharding Service as well as the current partition assignment. The GUI is updated with this information and the user can see how after changing the request load, the imbalance across caches moves towards a more balanced load with an updated assignment.

IV. REBALANCING ALGORITHM

Periodically, the ShardingService will aggregate the most recent metrics for each cache and analyze them to see if there is a significant imbalance. If the load needs to be rebalanced, we use our rebalancing algorithm, which will repartition the keyspace and reassign slices to new caches so that the load is theoretically balanced across all nodes, while trying to minimize key churn. The main research of this project went into developing the rebalancing algorithm, which uses what we call a Surplus/Deficit approach to redistributing the slices to each cache. Recall that each operation on a key carries a cost, which is expressed in milliseconds for how long it takes to execute. We therefore define a cache's heat to be the sum of the costs of operations that it executes within an interval of time. We illustrate our rebalancing algorithm with the following example.

Fig. 2. Cache imbalance detection

In this example, we have four caches and the heat of each cache. The first step is checking for a significant load imbalance based on the heats for each cache collected in the past cycle, which in our case is five seconds. In the Fig. 2, the average heat for a cache is 250. We use max/mean, and mean/min as the metrics to detect an imbalance in load. We use max/mean to detect if a node is particularly hot and we use mean/min to detect if a node is particularly cool. We have a threshold value for each of these metrics and if either metric crosses their threshold, we know there is a significant load imbalance. Based on empirical testing, we have computed the max imbalance threshold to be 1.25 and min imbalance threshold to be 0.75. We arrived at these two thresholds through testing various cases during which we cannot achieve better balance and hence we should not waste runtime by repeatedly calling the rebalancing method

if there is no benefit. In the case of Fig. 2, we can see the max imbalance = max/mean = 400/250 = 1.75 $>$ max threshold, so we run the rebalancing algorithm to help fix the imbalance.

The first step of the rebalancing algorithm is to group the caches into overheated caches and cool caches compared to the average heat. We say that an overheated cache has a surplus of heat, while a cool cache has a deficit of heat. The second step of rebalancing is going through the caches with surplus heat and removing some slices so that the expected heat of each cache is now within a bound of the average heat. These removed slices are added to what is called the surplus pool. This process is illustrated in Fig. 3.

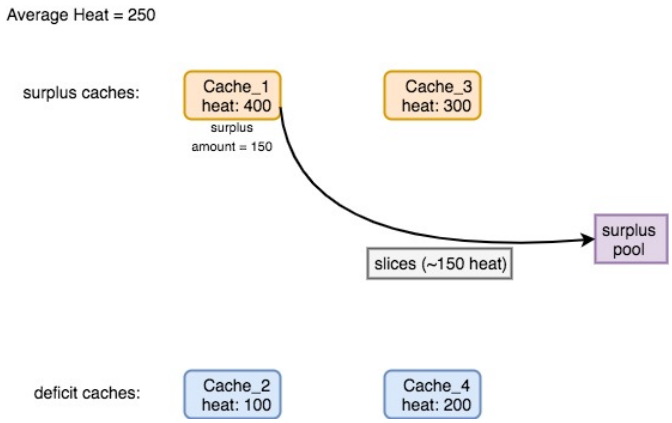

Fig. 3. Moving slices from a hot cache to the surplus pool

One of the main areas of research for this project was finding the slices to remove from a cache such that the removed heat equals the surplus heat of the cache. Recall that each cache sends the Sharding Service per-slice metrics so the rebalancing algorithm is aware of how much heat each slice in a cache has. When deciding which slices to remove from a hot cache, we first try to find if there exists some combination of slices in the cache whose heats sum up to the surplus amount. If we find such a subset of slices, we directly move these slices to the surplus pool and the current cache will be theoretically balanced.

If we cannot find such a subset of slices, the next method is splitting slices. We first sort the list of slices in descending order of heat. We continually add each slice to the surplus pool, provided that removing the slice wouldn't cause the cache to now have less heat than the average. When we get to the point where we have to remove a slice because we are still over the average heat, but can't remove the whole slice because it would put us under the average, we split the current slice. We originally thought to simply split the slice in half, but later came up with a better way to split a slice to obtain the desired heat. To determine how to split the slice, meaning how big each resulting slice should be, we have to introduce a new metric. We compute the metric, heat-per-key, which is obtained by dividing a slice's heat by the number of keys in the slice. Using this metric, we can estimate how many keys should be in the new slice going to the surplus pool and how many should stay with the original cache, based on how much surplus heat still remains.

There is a special case to the split procedure when we arrive at a hot slice that must be split but only consists of one key. This hot key problem is addressed by increasing the spread of the single key. Specifically, in addition to being served by the original cache, the key will be added to the surplus pool for another cache to serve.

The next step in the rebalancing algorithm looks at the cool caches. Each cool cache has a heat deficit, the average cache heat minus its heat. For a given cool cache, we want to find a subset of the slices in the surplus pool whose heats sum to the heat deficit. It turns out that this problem of finding which slices to move from the surplus pool to cool cache can be solved in exactly the same way as how we solved the problem of choosing which slices to move from a hot cache to the surplus pool. The only difference is that now we are filling a heat deficit rather than removing for a heat surplus. Once again, this procedure may involve splitting a slice that is in the surplus pool to bring the cool cache's theoretical heat within a bound of the average cache heat.

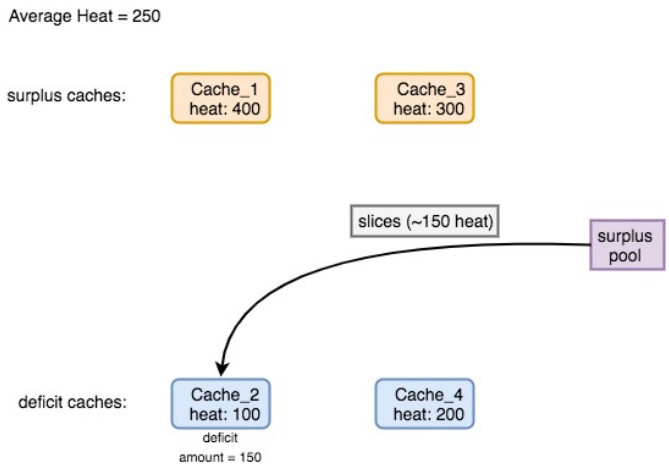

Fig. 4. Moving slices from the surplus pool to a cool cache

Once we finish the redistribution of slices to each cache and have theoretically balanced the cache loads, the last step is to merge slices to reduce unnecessary fragmentation of the key space. Within a given cache, we can merge two adjacent slices provided that the spread of each slice is equal to one. For the slices with a spread greater than one, on the next cycle of rebalancing, we first see if we can reduce the spread of a slice. This takes place before the Surplus/Deficit part of the algorithm.

It is important to note how the Surplus/Deficit approach of the algorithm helps to reduce churn. It minimizes churn by only moving the slices necessary to achieve a balanced load. Specifically, when we are looking at a hot cache, we only move to the surplus pool those slices which must be removed to bring the hot cache's heat down to average.

V. TESTING THE IMPLEMENTATION AND RESULTS

Once we implemented components of the project, we needed to verify the correctness of our implementation. We first created various unit tests to verify the correctness of different parts of our implementation. For example, we wrote a test that demonstrated that KVCache measures its load and slice heats correctly.

After unit testing our implementation, we ran our code with various request loads, using the command line API to generate specific request loads. We saw that our algorithm correctly updates the assignment as expected and in the correct number of cycles. Additionally, with a constant request load, we see the load balance reach a steady state after a given number of cycles. We also tested the algorithm by modifying the request load dynamically by creating and deleting Client Applications, and the load balancer adapts as expected.

VI. FUTURE WORK

For this project, we designed an algorithm that tries to minimize churn when updating an assignment with the Surplus/Deficit approach. However, in the Slicer paper, they describe using a weighted move algorithm that places a hard upper-limit on the amount of churn that can happen in each remapping stage. One next step would be to try augmenting our algorithm with a sort of upper limit on churn. When considering that all our strategies to balance the load on a given cycle are based on heuristics like heat-per-key, even if our algorithm can perfectly balance the theoretical load, it is not guaranteed that such balance is actually obtained. This is partly because heat-per-key is an approximation ignoring the fact that a slice may be hot only because of a single or few keys. Thus, when we split the slice in proportion to how much heat to take away, the slice we are left with may not have the intended heat. Additionally, our algorithm tries to balance the request load across caches for the next cycle, based on the request load for the previous cycle. While the request load from one cycle is likely a good predictor of the next cycle's request load, this is still a prediction. It could be that the request load drastically changes each cycle, in which case, theoretically balancing the load after one cycle doesn't mean that the load will actually be balanced with the new request load. This is especially significant if rebalancing the load requires a lot of churn. Practically, it doesn't make sense to remap a lot of the key space to achieve a balanced load for the previous cycle if the next cycle's request load is just as imbalanced because the request load changed. In this case, balancing the load may not be worth all the cache misses caused by the high amount of churn.

Another area we would like to investigate more is how to best deal with a hot slice. There are two options we use to deal with a hot slice: splitting it and increasing its spread so more caches serve it. However, we almost always use split unless the slice is a single key, in which case we are forced to increase the spread. The preference to always split hot slices can result in more fragmentation of the key space, making the assignment information larger. If instead we adapted the algorithm to better modify the spread of a slice, we could achieve a less fragmented key space and more compact partition mappings. One downside of using spread instead of split is that you can only divide a slice into two slices with theoretically the same heat. On the other hand with splitting, you can divide a slice proportional to the desired Surplus/Deficit heat. One extension to increasing the spread of a key would be to support increasing the spread by more than just one node at a time.

Because we focused on the load balancing algorithm itself and used spread > 1 at times, we did not implement a consistent assignment as Slicer can do. Recall, a consistent assignment, as defined in the Slicer paper, is where a given slice can be served by at most one cache at a time. This property is necessary if your application has a consistency requirement. If we were to extend our implementation to support this, we would have to never allow a slice to have spread > 1 and would have to modify the process of propagating an updated assignment.

In addition to modifying the algorithm, it is important that we apply our load balancing work to a real sharded service. This way, we can measure the load balancing capability of our algorithm and heuristics in a real world setting instead of with an artificial request load.

VII. CONCLUSIONS

For this project, we sought to design a load balancing algorithm general enough to work well with any sharded service. We implemented and tested our algorithm with a generic key value cache and a test harness capable of generating any arbitrary request load. The results of our testing confirm to us that the algorithm does in fact meet the project's initial goals.

As mentioned in the future work section, one issue that needs to be further considered is churn. Specifically, we could try adding an allowance for how many keys can be remapped in a given cycle to limit unnecessary churn. In addition, we could reduce churn by having a wider bound for what is considered balanced. In other words, we may save the cost of rebalancing a request load if it is good enough, especially considering the request load could change in the next cycle and regardless what we do now, it will be imbalanced.

However, given the assumptions made for this project, we believe that our load balancing algorithm will work well with a wide range of sharded services, especially if the request load doesn't drastically change between two cycles. We believe that in the common case, our algorithm both efficiently balances the request load among nodes and minimizes unnecessary key churn.

REFERENCES

[1] A. Adya, D. Myers, J. Howell, J. Elson, C. Meek, V. Khemani, S. Fulger, P. Gu, L. Bhuvanagiri, J. Hunter, R. Peon, L. Kai, A. Shraer, A. Merchant, and K. Lev-Ari. Slicer: Auto-sharding for Datacenter Applications. In Proceedings of the USENIX Conference on Operating Systems Design and Implementation, OSDI16, pages 739753, Berkeley, CA, USA, 2016. USENIX Association.

[2] Bruce M. Maggs , Ramesh K. Sitaraman, Algorithmic Nuggets in Content Delivery, ACM SIGCOMM Computer Communication Review, v.45 n.3, July 2015

[3] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazons highly available key-value store. In Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles (2007), ACM Press New York, NY, USA, pp. 205220.