Chopper: Dynamic Load Balancing

Kiet Pham
kap036@ucsd.edu

Chi Cheng Lee
ccl010@ucsd.edu

Heidi Cheng
h8cheng@ucsd.edu

Abstract

Managing load is crucial to running applications at scale efficiently. Many applications implement custom load balancing components which take considerable time to develop and test–incurring costs and delaying time to production. Chopper is a general purpose sharder, designed to work with different types of applications and reduce the burden of scalability for application development. Chopper dynamically load balances based on load metrics monitored on servers. It aims to decrease load imbalance with little key churn and minimal overhead.

For this report, we present the design of Chopper, largely inspired by Google's Slicer [7]. We discuss the implementation and demonstrate its efficacy on 2 simulated quintessential types of applications–a key-value based application and a compute-intensive application–with various key numbers and distributions. We show that even with few servers and at rates of thousands of requests per second, Chopper's load balancing can achieve over 100x improvement in latencies and more than 60% increase in throughput over consistent hashing for some scenarios.

1 Introduction

Load balancing and sharding are established mechanisms to deal with the challenges of scalability in large scale applications. In principle, good load balancing should ensure efficient utilization of available resources, avoid overload and server crashes, and allow for low latency and high throughput. Ultimately, this results in lower maintenance and labour expenses, as well as less resources required and low waste. Correspondingly, this also reduces the carbon footprint of applications (especially those running in data centers), especially since the low utilization on some machines, which is one consequence of improper load balancing, would be extremely power inefficient and may even be unnecessary [8]. Accounting for load balancing will result in better peak capacity planning.

Nowadays, it is often the case that a single machine's resources will not be sufficient in handling the load when an application goes to production. Therefore, modern applications rely on horizontal scalability to meet their performance goals. However, horizontal scaling necessitates the challenge of the aforementioned load balancing–distributing work in a logical and/or efficient way. For applications where requests handling is interchangable at any server, this is important, particularly to avoid server overload. For applications where affinity is important for efficiency, careful requests routing is crucial for performance and cost. Generally, applications would like to be able to handle loads that are dynamic and skewed. Traditional forms of load balancing like consistent hashing offer statistically uniform distribution of mappings but this is not sufficient for many workloads with high peaks and they don't offer guarantees beyond probabilistic bounds. However, the other option of having to create a per-application sharder is expensive and time consuming, and will require extensive effort to test and tune.

Chopper's design allows it to easily integrate with existing applications to provide these load balancing needs with minimal additional effort. Applications simply need to provide a key for sharding, and Chopper takes care of routing, load monitoring, and dynamic balancing. It separates the centralized control plane from the distributed data plane such that load balancing decision making do not block hot data paths. It allows for many local decisions, not dissimilar to the case for static hashing, but has more flexibility in assignments (and reassignments) of key ranges to servers. Its design accounts for server affinity, key churn, and load skews.

This paper will focus on 2 elements:

- Chopper's design and implementation
- Evaluation of its performance on various simulated

workloads and applications

Section 2 reviews popular types of applications and how they could use Chopper. Section 3 covers the overview of Chopper and its sharding model. Section 4 presents the high level architecture and components of Chopper. Section 5 discusses implementation details. We explain our evaluation plan, components for load testing, and review the performance results in Section 6. We recognize limitations and explore areas for improvement and potential future work in section 7. Finally, section 8 summarizes our results with Chopper.

2 Applications

Many types of applications can benefit from a general purpose dynamic load balancer like Chopper. However, Chopper's current state of development does not support applications requiring strong consistency semantics at the server layer for load balancing; applications with weak/eventual consistency semantics or those that rely on another layer (e.g. an underlying storage system) for strong consistency, however, can use Chopper with full correctness. This restriction is not essential to Chopper's efficient implementation but is a consequence of time constraints–Chopper can be extended to service many more types of applications in the future if needed.

With increasing usage and popularity of the World Wide Web, there has been increasing numbers of web applications needed to meet the demands for content accessed online. Consequently, many applications use DNS servers and other type of in-memory cache servers.

Other applications involve servers for fairly CPU intensive tasks (and perhaps data intensive too) or managing loads that a user would prefer not to have to handle on the client side. These applications could be RPC servers frequently invoked to run certain procedures for example. Regardless, it is not rare to find such types of applications that are fairly computationally intensive, yet not necessarily requiring a HPC setup. In fact, many data center applications are of this category, especially with the prevalence of stream/batch processing and dataflow types of applications.

For these reasons, we have chosen 2 types of simulated applications to study with Chopper. Section 6 will elaborate on these applications.

3 Overview

3.1 Sharding Model

Applications can provide a key to Chopper as the basis for sharding. This flexibility allows the applications to make informed decisions regarding how they would like their work to be divided. For example, keys could be user IDs or URLs if the application benefits from affinity to state stored for those users or content/resource served by that URL respectively.

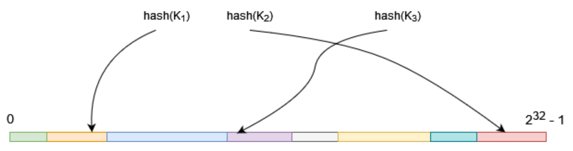

Figure 1: Chopping keyspace into slices

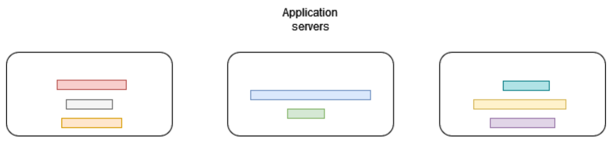

Figure 2: Assignment of slices to servers

Every application key is mapped to a 32-bit *slice key*. Since this mapping will be required on the data path (see section 5), it is important to select a fast function–we employed FarmHash's platform-independent fingerprint32 hash function [9]. This design decision allows Chopper to operate on slice key ranges (slices) as shown in figure 1 within the keyspace regardless of the application's need for application keys–there is no restriction on how many keys an application wishes to use.

A hash function allows many decisions to be made locally (as are done on the front ends in section 4.1), without having to bottleneck a centralized decision maker for every request. Moreover, a good hash function offers a natural degree of load balancing from the uniform distribution of hashed values (in this case, the slice keys). This is particularly useful in distributing hot keys across servers. To account for uneven distributions and dynamic shifts in load patterns, Chopper dynamically rebalances by changing the assignment of slices to servers. Servers may be assigned multiple slices of various lengths with no particular restriction on ordering as seen in figure 2. Slices can be moved, split, and merged as necessary to accommodate load balancing (see section 5.2).

3.2 Load balancing considerations

Ideally, close to "perfect" load balancing can be achieved if we can know the workload patterns in advance. However, for most applications it is impossible to predict future workloads accurately. One solution is to block a batch of requests first, compute the assignments and then distribute them evenly to application servers. Although workloads to all application servers will be relative balanced, the significant latency overhead is undesirable for most applications and unacceptable for applications that require real-time interaction or are user-facing. Moreover, this does not work well when load is high and there is a constant stream of requests. Another seemingly simple approach is to real-time monitor all application servers and assign requests to the server with the lowest utilization at that point. This may be reasonable for lower load applications which do not rely on state stored on the application servers for performance and are not sensitive to key churn. However, stateful applications would then have issues with mostly non-deterministic assignments and would perhaps need application servers to either replicate data to all servers or perform complicated quorum based techniques to achieve performance; in any case, the overhead of these non-generalizable solutions are high and the cost of real time monitoring is non-neglibile. Thus, it we concluded it would be wise to separate the data plane from the control plane as much as possible–these examples are to show the characteristics of a design we wish to avoid with Chopper. Inspired by Google's Slicer, we use a different components to achieve this goal: front ends for efficient local decision making about routing and data forwarding, and a controller to periodically monitor the workloads of application servers, updating these assignments and logically moving slices, which represented for a range of hashed keys, between servers.

4 Architecture

There are three main components in our dynamic load balancing system, application servers, front-end servers, and controller. To achieve dynamic load balancing, we designed a centralized controller to periodically monitor and balance the load among application servers.

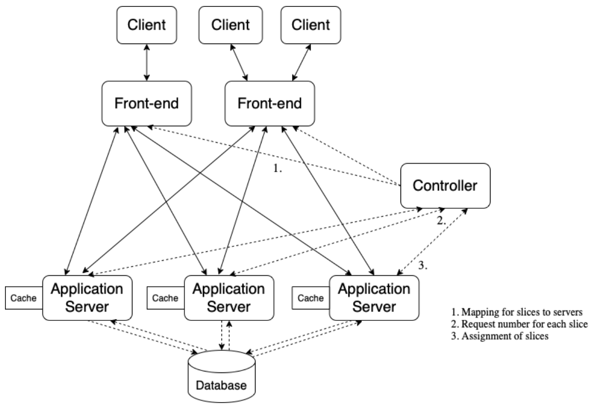

Figure 3: Architecture

Figure 3 shows the high level architecture of Chopper. The cache and Database components are specific to the key-value application we test, but the other components would still be present when Chopper is used with other types of applications in general.

4.1 Front-end Servers

Front-end servers are responsible for forwarding the requests from clients to the appropriately assigned application server and responding to clients after receiving the responses from application servers. Front-end servers redirect the requests based on the mapping of slices (hashed key ranges) to application servers–binary search is done on a data structure of sorted slice mappings to quickly determine the destination. These mappings are sent by the controller periodically. This means that front ends are able to efficiently do local decision making for request routing, and only need to briefly pause request handling when receiving a new mapping (assignment) from the controller.

4.2 Application Servers

Application servers handle the requests and generate responses. They receive requests forwarded by front ends. Chopper's integration into application servers also enforces each application server check each request to make sure it is still responsible for handling the corresponding slice key; moreover, request counts for each slice it is handling is tracked here as well, and a binary search based algorithm is used on a data structure of sorted slices assigned to the server to efficiently complete these functions.

Various applications can integrate with Chopper and run on the application servers. For the purposes of evaluation, we work with two kinds of applications. One is a key-value based application and the other runs longer computations. For the key-value based application, application servers serve as in-memory caches by running local Redis servers and we use DynamoDB as the underlying storage layer / database. The other application runs computations directly on the application servers and does not need additional infrastructure.

4.3 Controller

The centralized controller monitors the load of each application server, which are the aggregate request rates across slices assigned to the server. It carries out the load balancing algorithm, described in more detail in section 5, based on this information to determine the new assignment and sends them to the front ends as well as appropriate assignment subsets to application servers. For this purpose, the controller maintains persistent connections with both front ends and application servers. In terms of control data flow, the controller receives per slice request rates from application servers and updates both front ends and application servers with slice mapping information. These events happen periodically to achieve dynamic load balancing with low overhead–request counts are refreshed for each load balancing period to only measure the relevant load information for the each new mapping assignment.

5 Implementation Details

5.1 Tools and Frameworks

We used a variety of existing libraries and frameworks to produce the logical functionality of Chopper. The choices are partly based on research but mainly based on experience and time concerns–nothing prevents the same logical implementation using different frameworks. We would like to noe the main high-level frameworks we used. To simplify development, we decided to use Node.js as the runtime environment for our servers. Accordingling, we utilize the following:

- Express [10] as a minimalist framework to handle HTTP requests on servers.
- Axios [4] to send requests to communicate between components.
- The cluster module [6] to run multiple Node.js processes to maximise utilization of cores available on servers, since Node.js's architecture is single threaded.
- Cluster-hub [2] for inter-process communication to have a consistent view between workers.

Other notable libraries and technologies we used especially for applications and load testing include Redis, DynamoDB, and wrk2. We discuss this further in section 6.

5.2 Load Balancing

To dynamically balance loads, an algorithm should aim to reduce the load on hot servers and increase the load on cold servers, which can be achieved by reassigning slices handled by application servers. Chopper has three phases to produce the new assignments, which are *merge*, *move*, and *split* and executes them in that order. We define imbalance as $\frac{maxServerLoad}{meanServerLoad}$. Although moving as many slices as needed to achieve minimal imbalance sounds desirable, the key churn costs and overhead created by assignment changes for application servers should be considered.

5.2.1 Key Churn

Chopper sets an upper bound on *key churn* to constrain the cost incurred (particularly in latency) by moving slices from one server to another. We define the amount (key range lengths) of moved slices as key churn and use each slice's key churn to approximate the cost for moving the slice. To more accurately estimate the cost of moving a slice, an more complex mechanism to monitor load within segments in a slice (or other variable/fixed/proportional granularities) may be pursued in the future.

5.2.2 Merge

To prevent the likelihood of having too many slices as well as avoid unnecessarily cold slices, Chopper merges adjacent cool slices. The threshold for categorizing cool slices is mean slice load. Since slices adjacent in the keyspace may be currently assigned to different application servers, we impose limits on key churn, which is currently 20% of the keyspace based on experiment tuning, for merging slices to constrain merge overhead. The merged slice will get assigned the combined load from the slices that merged to form it for use in further steps of the algorithm. The adjacent slices, which can be two or more slices, will merge if they meet the following conditions.

1. If the merge is between two servers, the merge should not exceed the key churn limit
2. The merged slice has load lower than mean slice load
3. The number of slices will not be smaller than the lower-bound of number of slices, which we used the amount of application servers as our lower-bound, to avoid idle server.

5.2.3 Move

As mentioned above, our strategy to balance load is moving slices from the hottest server to the coolest server. A logical way to cool the hottest server is to remove its hottest assigned slice, and move it to the coldest server. However, if the coolest server will become even hotter than the old hottest server after the move, the move will not execute–the hottest slice remains in the current hottest server and will likely be split later, and the second hottest slice will be considered for moving. The controller will try to move as slices as it can from hotter servers to cooler servers until it hits the key churn limit. We separate key churn limit for move from that for merge since we do not want one of the two operations to use up all the key churn quota. The following are the steps for move.

1. Sort slices in servers by the request number received in this period
2. Get the current hottest server and the coolest server at this point
3. If the load ratio between the hottest server and the coolest server is smaller than a threshold, which is 1.25 in our algorithm, the controller breaks out of the move phase.
4. Choose which slice to be moved
   (a) Start from the hottest slice in the server
   (b) If the coolest server will not become the hottest server after slice moved, the slice is chosen for move.
   (c) If moving the slice will create greater imbalance, we skip this slice and proceed to the next slice in the sorted slices array and do the previous check again.
5. If this move will not exceed limited key churn, move will be executed. Or controller will stop moving any slice.
6. Update loads of application servers after move, and repeat from step 2.

Note, the hottest or the coolest server may be same ones as in a previous iteration within a round of the algorithm. However, we prevent moving slices from the previous coolest servers to avoid moving slices back-and-forth between servers. Also, if the hottest server has no other slice rather than the hottest slice to be moved, we will move to deal with the second hottest server and balance the hottest server in the next iteration after the hottest slice in the hottest server has been split.

5.2.4 Split

Chopper splits the slices which have load higher than the mean slice load into two slices. We split a slice from the middle point of the range; that is to say, slice with range (a, b] will be split to two slices with range (a, (b-a)/2 + a] and ((b-a)/2 + a, b] respectively. The slices after split will stay in the assigned server. The resulting 2 slices from the split each get assigned half of the original slice's load.

6 Evaluation

We evaluate Chopper's performance by comparing the experiment results of load imbalance, latency distributions, and throughput with those for a static/consistent hashing version of load balancing. For these tests, we have the controller execute its algorithm every 5 seconds to scale down to the duration of fairly short tests, which are done for time and cost efficiency purposes.

6.1 Tested Applications

As discussed in section 2 and 4.2, we evaluated Chopper on two applications. The first is an in-memory cache application particularly to serve key-value requests and uses DynamoDB as the underlying store. Writes update data in DynamoDB as well as the Redis cache of the application server handling the request, and reads uses Redis as a look-aside cache–only upon a cache miss would the application read from DynamoDB. To achieve eventual consistency semantics for this application as well as guarantee that we account for key churn costs of cache misses, we set Redis entries to expire in 10 seconds from the last access. For this application, we use the requested "key" for the key-value request as the application key.

The other application runs longer computation to simulate workloads that require reasonable CPU times. Currently, we run an order $O(N^2)$ summation operation using looping where N is around 1000. At first glance, this may not seem like much but handling thousands of requests per second, each requiring 6 orders of magnitude (~1M) of operations, on top of processing network requests and responses can easily drain CPU availability. Moreover, a reasonable N allows us to control utilization better during experiments by modifying request rates. For this application, there is no particular meaning associated with the application key we used for simulation besides the fact that it simulates some sort of sharding key specified by an application, likely one that relies on state to run / optimize their requested runs.

6.2 Load Generation and Probabilistic Distributions of keys

To evaluate Chopper's performance on various types of workload patterns, we implemented a workload distributions generator. We considered using tools like YCSB [3] for driving load with various workload patterns but felt it was not as customizable and believed we could benefit from our own workload pattern generator, which we could also understand better and tailor to our needs. Since Chopper works with the abstraction of the application key for sharding, when it comes to load balancing, the most important point would be the (probabilistic) distribution of keys. This was our main focus for our workload pattern generator, which would generate the data (application keys) used for experiments.

Firstly, we would like to have different phases in a single experiment. This is important to evaluate Chopper on dynamically changing loads, since variations within a probability distribution alone is not a sufficient test for that. For example, if there are 3 phases in an experiment, the load of the first phase can be pretty low, and the load can become more and more intensive in the later phases. Also, the key distributions in the later phases can be very different. Suppose that we have 5 keys for a specific experiment, the second phase can have intensive loads on the first and the second key, and the third phase can have intensive loads on the third and the fourth keys. The set of possible keys need not be the same across phases. Our generator allows customization of different durations and request rates for different phases to generate keys to be used for our experiment.

Moreover, we designed the generator to support various kinds of key distributions, even customized ones. We can specify distributions like the Gaussian Distribution, the Poisson distribution, the Uniform distribution etc. We can also specify the parameters for each distribution in the function or provide our own customized distributions.

Additionally, our generator also creates the random keys and values to be used as well as specify length bounds for them. To simulate workloads with fairly small values for the key-value application for example, we could modify this parameter. On the other hand, we could set this to be a sizable number that can significantly affect the performance of the program. Lastly, our program allows easy customization of get / set ratios for key value requests. This is important because the load incurred by get and set operations are different for key value requests (generally sets are more intensive) and allows us to use a higher get rate to more accurately simulate the purpose of an in-memory cache application.

Once the workload pattern is generated, we need a way to actually drive and generate requests for our load testing experiments. Our initial attempts used artillery v2 [5] but its payload reading mechanism was not performing as advertised. Although artillery v1 has the mechanisms to use our generated data correctly, artillery in general was not the most efficient load driver since it is designed to do various types of testing (e.g. smoke testing) besides load tests and has unnecessary built in features. Eventually, we moved to wrk2 [1], which is able to drive significantly higher loads and allows customized metrics collection and payload injection using Lua scripts, ensuring that load driving will not be a bottleneck in our experiments.

6.3 Experiment Setup

All machines are run on Amazon Web Services (AWS). We use two m5zn.xlarge instances for front ends and three t3.small instances for application servers. The load is driven by another m5zn.xlarge instance. This setup configuration guarantees that the bottleneck will not be in the load driver or the front ends. All machines run on AWS's us-west-2 region. The load driver distributes requests evenly to both front ends–although we would not be bottlenecked if we only used one with this setup, it is important to verify our implementation works for multiple front ends.

6.4 Results

We present the results of some insightful experiments we conducted. Most other experiments not included here exhibit similar outcome behaviours to the results of one of the following experiments discussed.

6.4.1 Key-Value In-memory Cache Application

All tests were run with 200 concurrent connections for 360 seconds (6 minutes). Around 80% of requests are get/read requests, and 20% are writes. All 3 app servers receive the same initial assignment for both consistent hashing and Chopper, with each app server being responsible for a continuous key range which is one third of the entire keyspace.

Our first experiment (figures 4 and 6) uses 10 keys, of which 3 were much hotter than the rest. There are three phases and each phase has a different set of 3 keys that were hot. This test aims to simulate scenarios where application uses few keys and a only a handful of them receive significant number of requests. We set a target of 3500 requests per second (req/s), which would put our server utilization from 80-100%. At close to max utilization, we would expect pretty high tail latencies since there will be cases when resources are too tied up to respond in an efficient manner. Consistent hashing was only able to achieve 3248.85 responses per second (res/s) while Chopper achieved 3494.10 res/s. The median latency was 32.35ms and 13.75ms for consistent hashing and Chopper respective. However, the latencies increased significantly at high percentiles for consistent hashing–notably, our algorithm achieved over 100 times smaller latencies compared to consistent hashing at the 90th percentile. There was little difference in imbalance until the final phase where consistent hashing's imbalance was consistently very high at over 2–for reference, in our setup, the maximum possible imbalance is 3. Moreover, our algorithm is able to quickly adapt to changing load patterns as indicated by the quick reduction in imbalance after phase changes (~10 seconds).

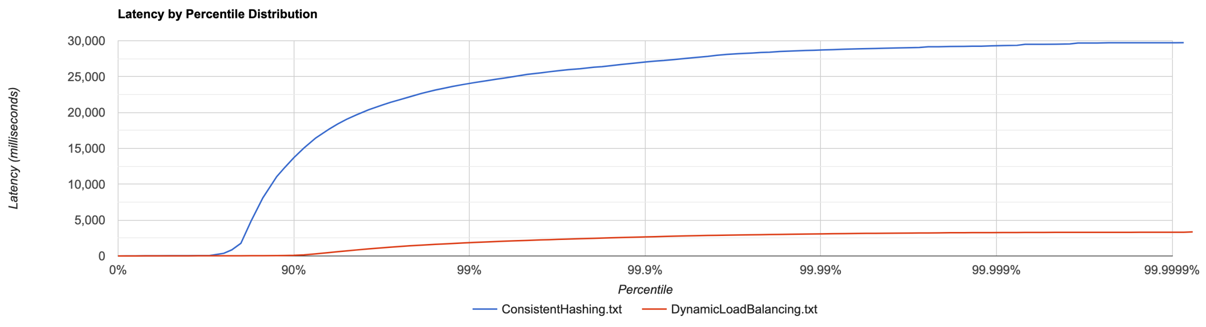

Figure 4: Latencies: 10 keys, 3500 req/s target test

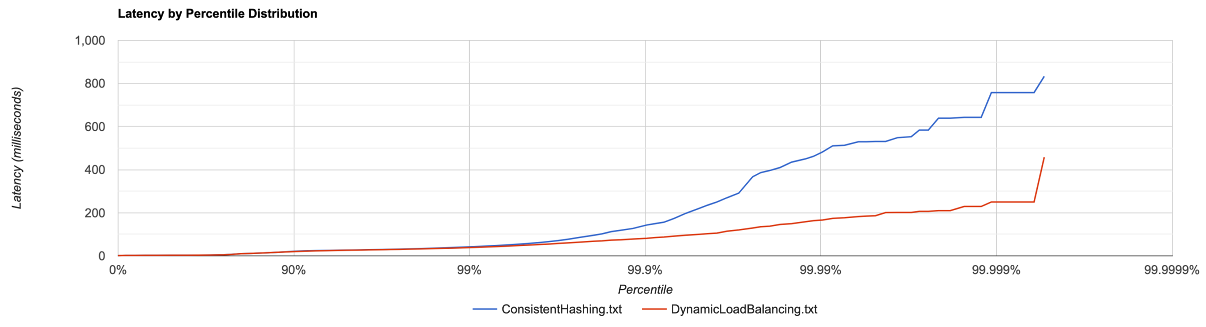

Figure 5: Latencies: 10 keys, 500 req/s target test

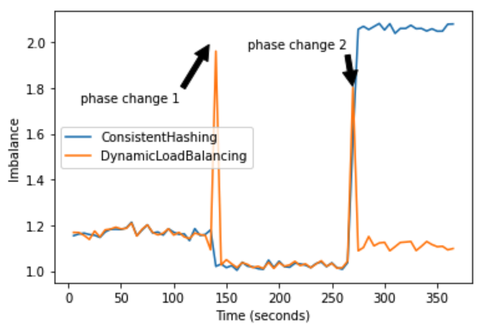

Figure 6: Imbalance: 10 keys, 3500 req/s target test

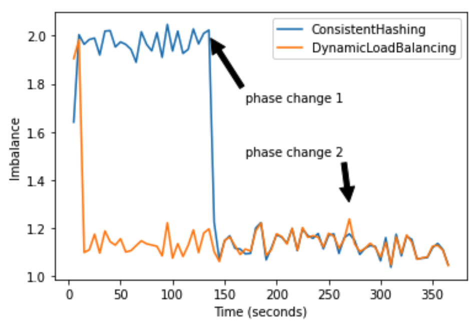

Figure 7: Imbalance: 10 keys, 500 req/s target test

The larger spike in imbalance for the first phase compared to consistent hashing is the result of Chopper having distributed a different mapping from the original that it thought was best for the first phase which does not fare as well for the second; however, Chopper's algorithm was quick to adapt accordingly.

The second test's configurations are identical to the first except for the request rate–we set a target of 500 req/s (figures 5 and 7). Since the load was significantly lower, the measured CPU utilization for our application servers was around 10-30%. Accordingly, although Chopper's results are noticeably better for the vast minority of requests, the results indicate little difference between the latencies observed for consistent hashing and Chopper's dynamic balancing scheme for most requests. Despite the much higher imbalance particularly in the first phase for consistent hashing, both schemes comfortably achieved a response rate of ~499.9 per second.

The third test (figures 8 and 10) uses 100 keys. There are also three phases: the first phase sees an exponential distribution of keys, the second a binomial distribution, and the last a normal distribution. The results here high-

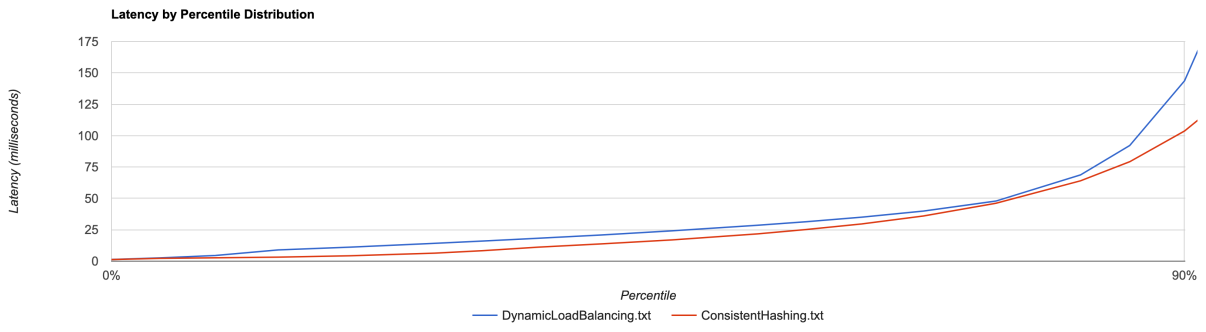

Figure 8: Latencies: 100 keys, 3500 req/s target test

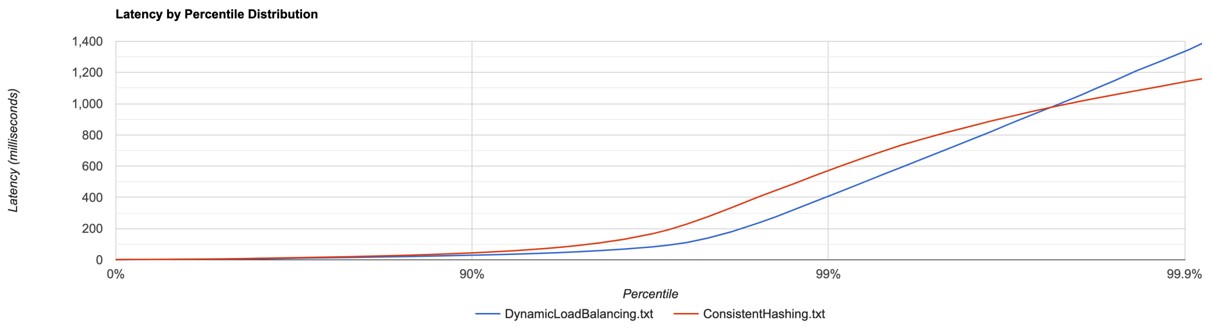

Figure 9: Latencies: 1000 keys, 3500 req/s target test

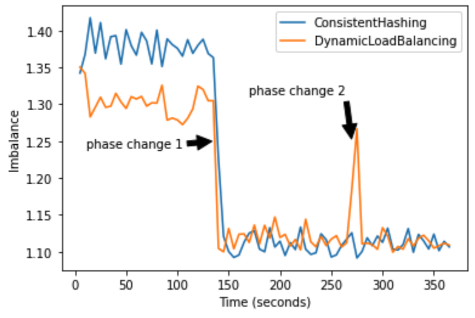

Figure 10: Imbalance: 100 keys, 3500 req/s target test

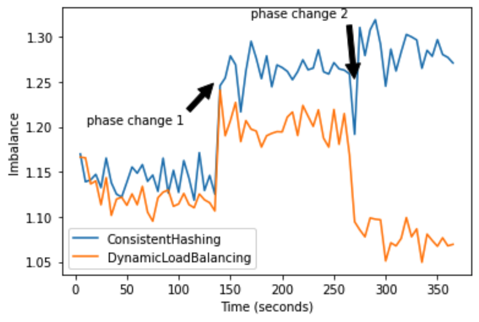

Figure 11: Imbalance: 1000 keys, 3500 req/s target test

light an interesting effect of hashing schemes–uniform distribution of mapped keys usually performs reasonably well when there is a large number of keys, that is relative to amount of servers. Moreover, consistent hashing actually performed better in terms of latencies across the board, despite having a higher imbalance particularly in the first phase.

Our final test (figures 9 and 11) was similar to the third test but with 1000 keys, but this time with 5 hot peaks. More specifically, we manually (programmatically) assigned high hotness to the 5 keys after specifying the probability distributions for each of the phases–our pattern generator is able to handle these customized distributions. The results are quite insightful–Chopper outperformed consistent hashing for the vast majority of requests but lose out at the 99.9 percentile in latency. We theorize that these requests reflect the cost of key churn after a new assignment is distributed by the controller since we would likely need to ready from DynamoDB every time a slice assignment is moved across / merged into another server. Since consistent hashing does not experience key churn, its high latency percentiles are

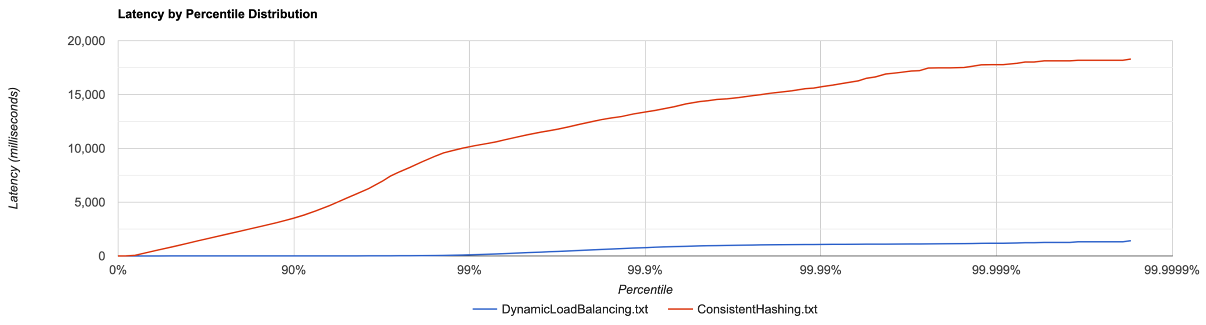

Figure 12: Latencies: Computations application–1600 req/s target test

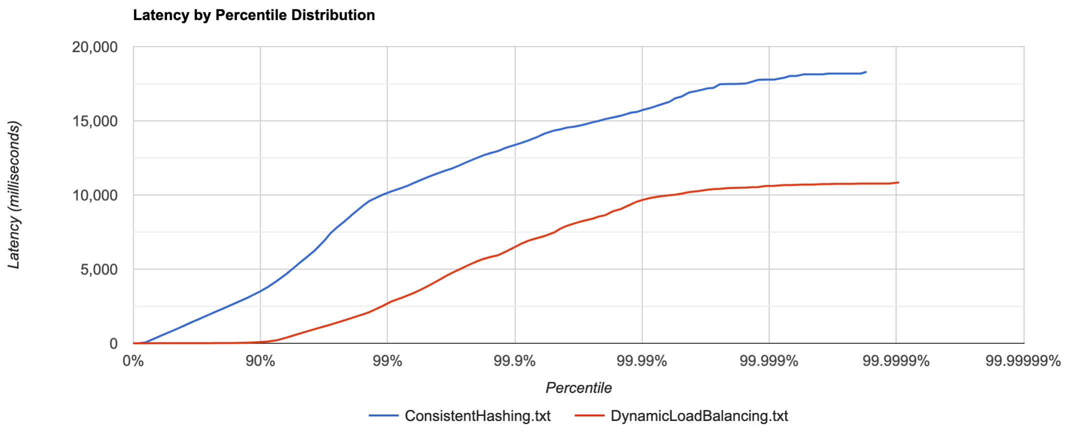

Figure 13: Latencies: Computations application–1600 vs 2600 req/s target

relatively more stable, despite having higher imbalance throughout all 3 phases, albeit the not too high overall. Both schemes achieved ~3497.8 res/sec with the target of 3500 req/s.

Chopper outperforms consistent hashing for most scenarios. These results suggest that we could improve Chopper by avoiding moves / merges of slices across servers when utilization is high but the current imbalance is sufficiently low, or if the key churn cost is fairly great–perhaps this could be specified by the application. However, we expect Chopper to outperform consistent hashing for most if not all cases if we work in environments with more servers (3 is quite small) or where servers are joining and leaving often–there will be a greater chance a server experiences much higher load than others.

6.4.2 Computationally Intensive Application

We run similar tests for this application. The main differences are in the request rates, as we set them based on the load utilization levels we wanted to test. As before, the test duration is 360 seconds. We focus on the 10 keys total with 3 hot keys test, as more computationally intensive applications can often have fairly low ratios of keys to servers.

With a target of 1600 req/s (figures 12 and 15) we were able to drive server loads to 80-100% for the consistent hashing case. The latencies are significantly greater for consistent hashing with the median latency being 819.2ms versus the 4.52ms for Chopper. This difference gets even greater at the 90th percentile, where Chopper saw a latency of 8.07ms vs 3.51 seconds for consistent hashing, over 400x that of Chopper's! The reason for this disparity becomes clear when observing the imbalance chart–consistent hashing had significantly greater imbalance throughout the runs. Still, both schemes achieved close to the target rate with Chopper getting 1599.16 res/s and consistent hashing 1597.27 res/s.

However, this was essentially at the throughput limit for the servers at high imbalance–we were not able to achieve over 1600 res/s for consistent hashing no matter how we set the target request rate. Crucially, compar-

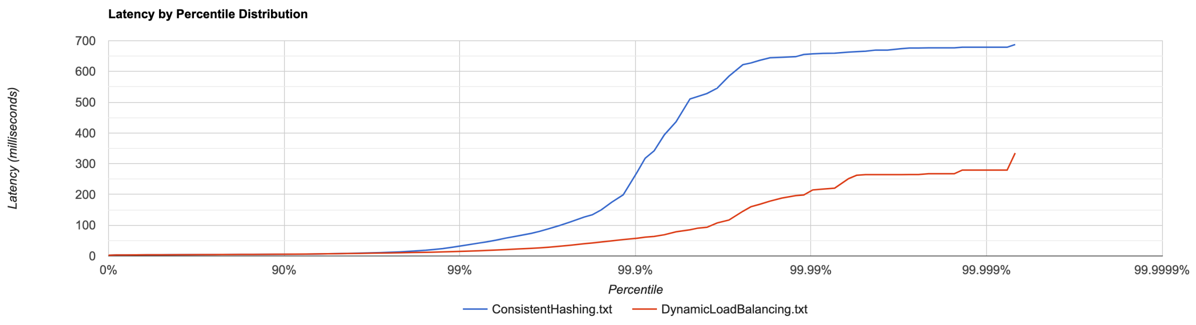

Figure 14: Latencies: Computations application–400 req/s target

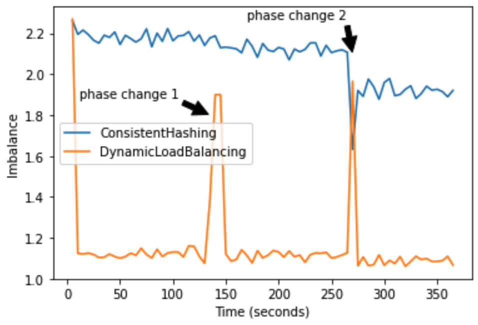

Figure 15: Imbalance: Computations application–1600 req/s target test

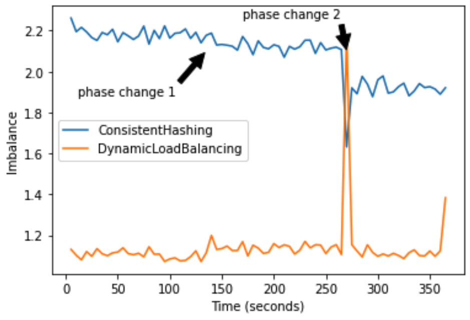

Figure 16: Imbalance: Computations application–1600 vs 2600 req/s target

ing this with a run where we drove higher load and using Chopper (figures 13 and 16), we were still able achieve sizably better latency results at this higher rate of 2600 req/s, over 1000 req/s higher! The response rate was 2598.43 res/s. In other words, we were able to increase max throughput by over 60% (up to 100% for some cases when we tried even higher rates) by just maintaining lower imbalance. Although, it must be acknowledged that due to the difference in request rates, we had to generate different key data for the experiments, and there is randomness involved in distributions. Regardless, the imbalance results are as expected, with Chopper achieving significantly lower results for both cases. Certainly, low imbalance in this case was a strong indication of better performance.

Finally, we ran the same test with a target request rate of 400 req/s (figures 14 and 17). The results are similar to that seen for the other application. Chopper achieved significantly better latencies only at higher percentiles, and there is less than 0.3ms difference between the two schemes for the first 90 percentiles of latencies. The median latencies were essentially identical with Chopper seeing 3.88ms and consistent hashing 3.89ms. Both achieved a perfect response rate of 400 res/s. Once more,

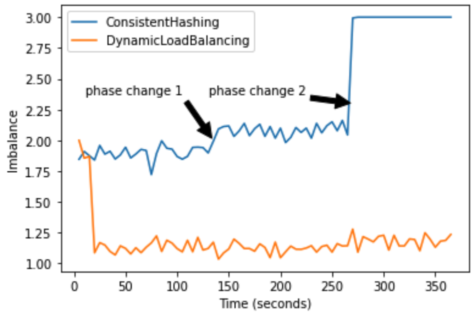

Figure 17: Imbalance: Computations application–400 req/s target

despite high imbalance, and in fact reaching the maximum imbalance of 3 for the final phase, consistent hashing performed decently in this case, but does fall short to its counterpart at high percentiles.

We also observed that $\frac{maxServerLoad}{minServerLoad}$ would give numbers easily in the 100s for some tests with consistent hashing, a consequence of one server receiving little to no load, which is never the case with Chopper–this is not highlighted as much with our definition of imbalance.

7 Limitations

Our current implementation of the controller only runs on one machine and may become a bottleneck if there are a large number of servers. For scalability and fault tolerance, we could enhance the controller design with replication over a consensus (e.g. Paxos) group, and increase fan-out for slice assignment distribution in the future.

We currently load balance on request rate, which works best if different request's with different application keys require similar amount of work. Although Slicer [7] showed that load balancing on request rate can achieve decent results, considering the CPU utilization of heterogeneous tasks might further improve the load balance in real workloads. Also, Chopper currently uses fixed thresholds for classifying cold slices, key churn limit, etc. However, these thresholds might not be suitable for all kinds of workloads. Dynamically tuning these thresholds based on feedback from collected metrics (e.g. imbalance) could address these weaknesses in the future.

Chopper currently does not handle joining, leaving, and crashing of application servers. We can also assign one slice to multiple application servers to achieve fault tolerance when asymmetric redundancy is supported and applications would like to trade off affinity for increased availability and scalability. Finally, we would like to perform tests at larger scale with more servers if time and budget allows, and evaluate it on real application workloads.

8 Conclusion

Chopper is a general purpose sharder that makes it easy for applications to benefit from well balanced server loads. It accounts for key churn and is designed to handle any type of workload and many types of applications.

Results from simulated workloads on applications show significant improvements in latencies and throughput for various server utilization levels especially for cases with high skews. Chopper adapts to workload pattern changes quickly and adds negligible overhead of control logic on applications. Future work on Chopper can extend its adaptability and support for even more customized load balancing on different metrics and support even more types of applications.

References

[1] wrk2. https://github.com/giltene/wrk2, 2014. Accessed: 2022-06-07.

[2] node-cluster-hub. https://github.com/sirian/node-cluster-hub, 2018. Accessed: 2022-06-07.

[3] Yahoo! cloud serving benchmark (ycsb). https://github.com/brianfrankcooper/YCSB/wiki, 2019. Accessed: 2022-06-07.

[4] Axios. https://axios-http.com/docs/intro, 2020. Accessed: 2022-06-07.

[5] Artillery.io — load smoke testing. https://www.artillery.io, 2022. Accessed: 2022-06-07.

[6] Node.js v18.3.0 documentation. https://nodejs.org/api/cluster.html, 2022. Accessed: 2022-06-07.

[7] ADYA, A. Slicer: Auto-sharding for datacenter applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)* (2016), Operating Systems Design and Implementation, USENIX, pp. 738–753.

[8] ANN STEFFORA MUTSCHLER. Improving energy and power efficiency in the data center. https://semiengineering.com/improving-energy-and-power-efficiency-in-the-data-center/, 2021. Accessed: 2022-06-07.

[9] GEOFF PIKE. Farmhash. https://github.com/google/farmhash, 2021. Accessed: 2022-06-07.

[10] TJ HOLOWAYCHUK. Express - node.js web application framework. https://expressjs.com, 2017. Accessed: 2022-06-07.