hw2.bib

Name: Patrick Johnson
Student ID: 5828453
Email: joh21451@umn.edu

Cloud Computing (CSCI 5980)

Coding Assignment 2: Enhancing Key-Value Store with Consistent Hashing Across Multiple Instances

---

Project Repository: `https://github.com/overdodactyl/key_value_server`

1 Overview

In this project, the previously designed key-value store was enhanced to support consistent hashing across multiple instances. In order to do this, small alterations were made to the existing API, the API was integrated with HAProxy to facilitate consistent hashing across server instances, and performance was evaluated using Locust.

2 Design Decisions and Justification

In the selection of technologies for the key-value store enhancement, Docker and HAProxy were chosen for their distinct advantages. Docker was utilized for its ability to encapsulate the key-value store in a container, providing a lightweight and portable environment that ensures consistency across different deployment platforms. This containerization facilitates easy scalability and simplifies deployment processes, which is essential in a distributed system setup. Moreover, Docker's widespread adoption and community support offer a wealth of resources for troubleshooting and optimization.

HAProxy was selected for its robust load balancing capabilities and its native support for consistent hashing, which is crucial for distributing requests efficiently across multiple server instances. Its proven reliability in high-availability environments, coupled with its performance in handling large volumes of traffic with minimal latency, made it an ideal choice. HAProxy's ability to parse and direct traffic based on URL parameters enabled the necessary routing of requests based on the key parameter, thereby aligning with the architectural requirements of the key-value store. The combination of these technologies underpins a system designed for resilience, efficiency, and scalability.

In order to analyze performance, Locust was used to simulate realistic user requests. The use of Locust allows more fine grained control of API requests, including simulating the number of users making requets, how quickly new users are added, and the wait time between requets for each user.

3 Challenges

One of the biggest challenges faced was getting the HAPorxy balencer to correctly point to the right server for a request. I originally tried to use the `uri` balencer, but this didn't work for a few reasons because the uri balencer does not take into account data passed into the body of the request. As such, all my requests had the same uri and were sent to the same server. This was fixed by making `key` a URL paramter. `value` was kept as an element passed in the body of the request. I had never used Locust or HAProxy before, so getting used to both of these tools also entailed a learning curve.

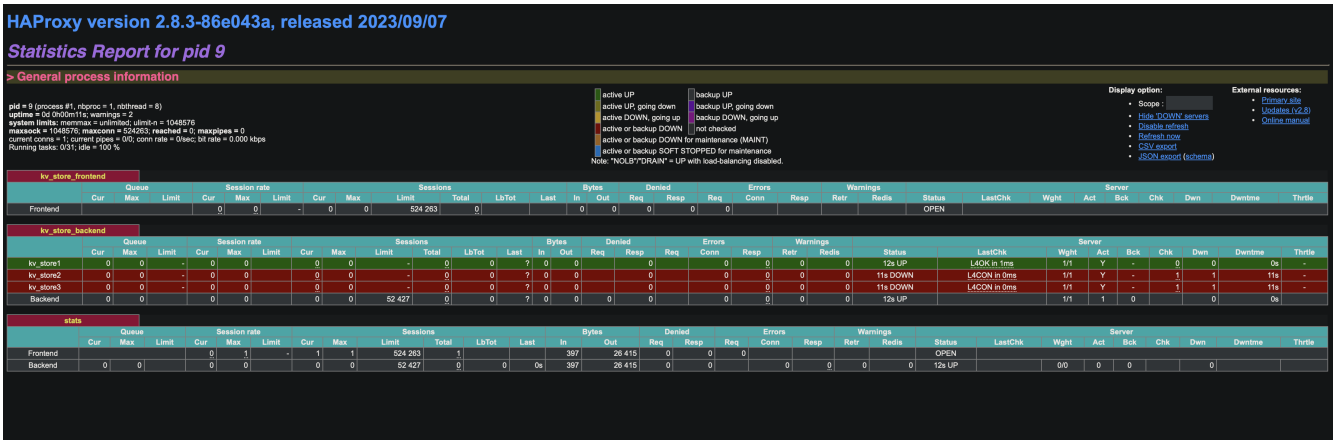

Figure 1: HAProxy with one server online

4 Testing

4.1 Key-Value Functionality

To test the functionality of the key-value server (both as a standalone instance and when running multiple), there is test code provided in `tests.py`. Noteably, the API was upgraded to retrun the server instance that the code was executed on (this can be set as an environment variable). While this may not be used in a production setting, it was implemented to help understand how making various changes (for example, in the balancer implementation) could impact where calls were being made. The test code also shows the count of calls sent to each instance:

```
KVStore1 received 38 requests.
KVStore2 received 28 requests.
KVStore3 received 34 requests.
Number of retrieval errors: 0
```

4.2 Multiple Instances

The testing page provided by HAProxy made it easy to test the implementation of multiple instances, including whether or not servers where online, where requets were being routed, the number of failed calls, etc. Figure 1 shows HAProxy after a single server has been brought online and no requets have been made.

Figure 2 shows HAProxy after 3 servers have been brought online and multiple requets have been made distributed across the servers.

4.3 Performance

Performance was evaluated via throughput and latency. In order to simulate real requets, Locust was used. The file `locustfile.py` contains the logic used to to generate queries. In short, random key/value pairs are created and `PUT` requets are made 10x more frequently than `GET` requests. Each user waits between .1 and .2 seconds between requests. Statistics are saved out every second. You can also specify the number of users (e.g., 1000) and how quickly new users are spawned (e.g., 5 every second). For each requets made, Locust reports summary statistics along with any failed requets. Fig 3 shows the dashboard after several different runs.

In order to compare the performance with 1-3 key-value servers, a locust run was executed with each number of servers. Between each trial, all docker images were completely shut down and restarted rather

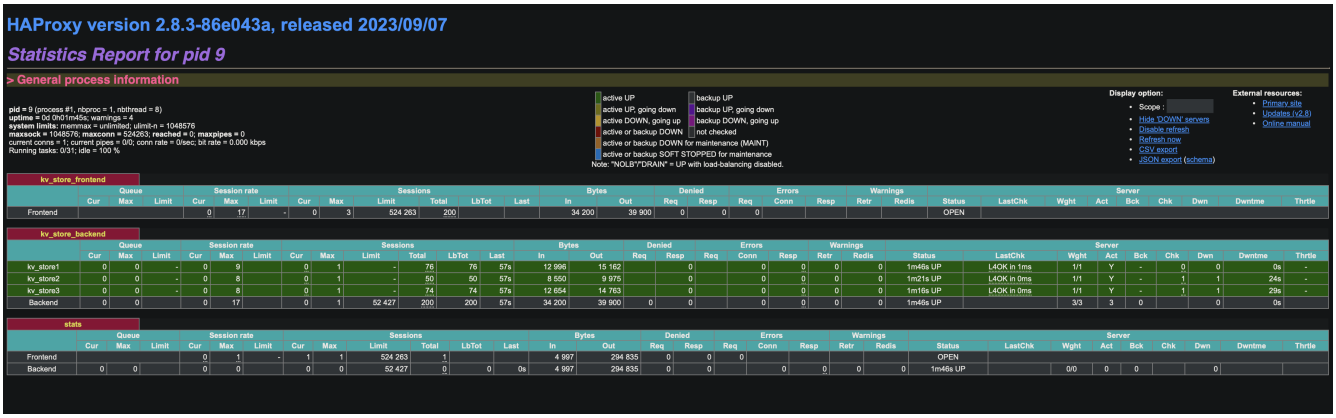

Figure 2: HAProxy with three servers online

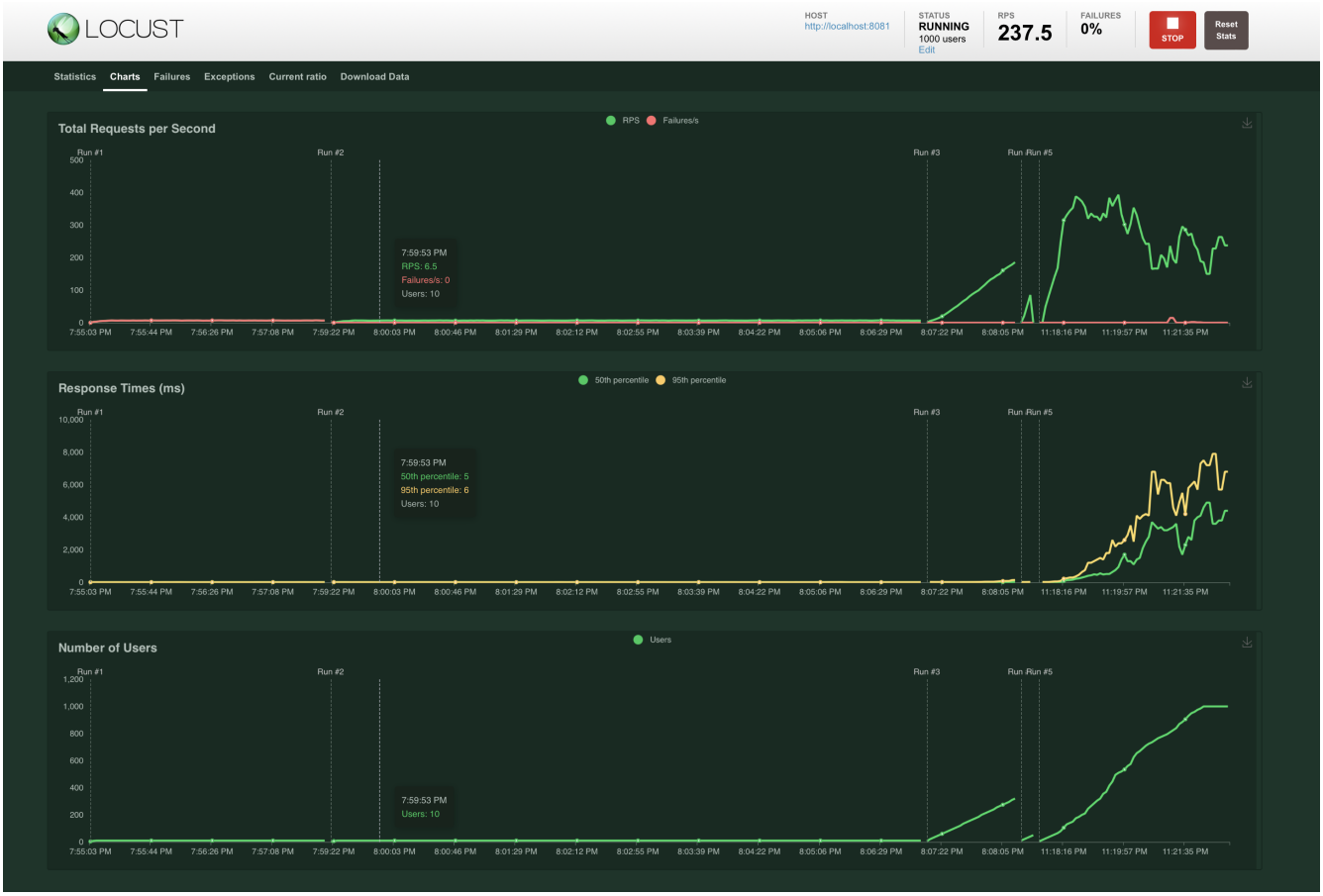

Figure 3: Locust performance dashboard

than simply adding new servers. This was done to avoid any bias from the first server having to maintain records and information from previous runs. The code for this performance is in `performance.sh`. After the completion of all runs, throughput was plotted against latency, and is shown in 4.

The plot can be summarized by the following observations:

1. Low Throughput Behavior: When the throughput is low (left-hand side of the graph), the single server setup exhibits the lowest latency. This indicates that when handling a smaller load, the single server is more efficient, likely due to the absence of inter-server communication and load balancing

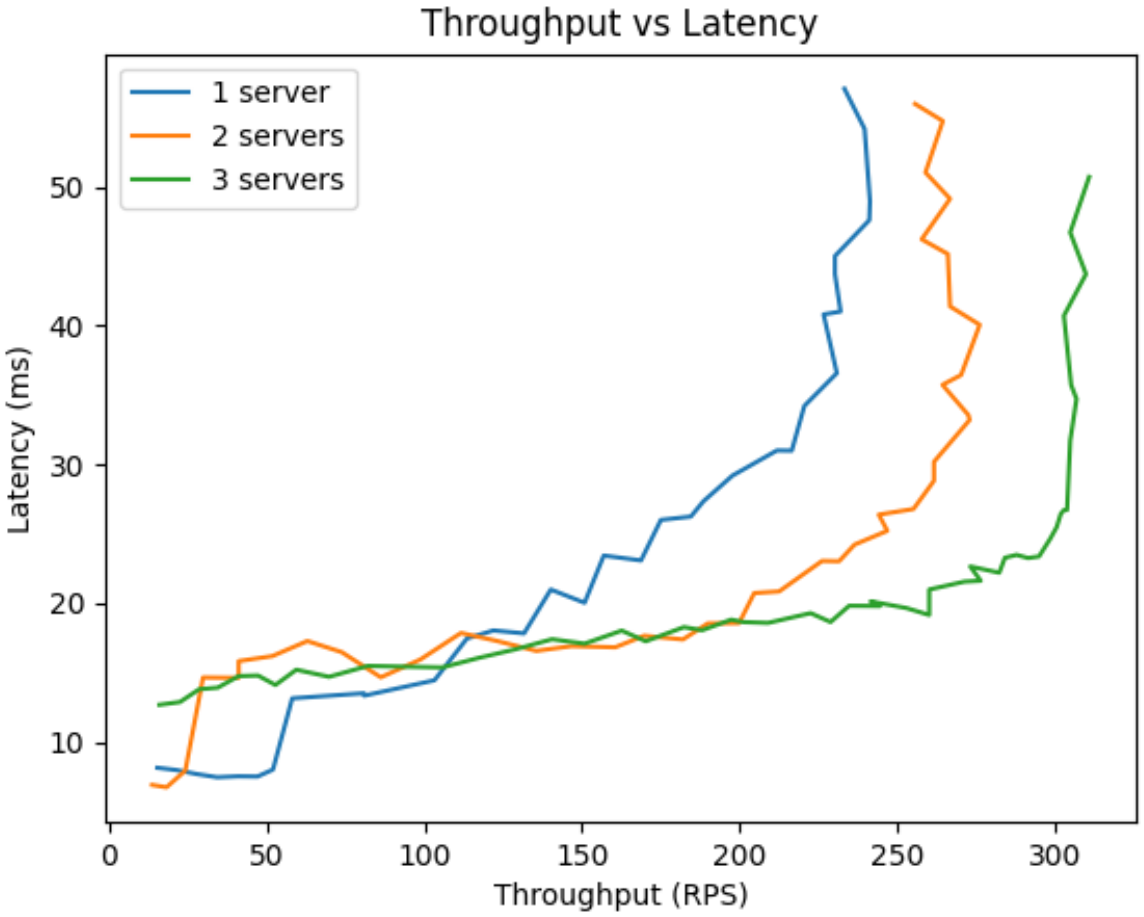

Figure 4: Key-Value Performance With Varying Instances

overhead that is present in multi-server configurations.

2. Latency Trend: As the throughput increases, the latency for the single server setup starts to rise significantly, surpassing the latency of both the 2-server and 3-server setups. This suggests that a single server becomes overwhelmed with a higher number of requests, thus taking longer to process each one.

3. Multi-server Efficiency: The 2-server and 3-server setups show a more gradual increase in latency compared to the single server. This behavior exemplifies the benefits of load balancing where the additional servers are able to manage increased request rates more effectively, keeping latency lower for longer as throughput increases.

4. Optimal Throughput Range: There's a middle range of throughput where the 2-server and 3-server setups appear to handle the increasing load without a substantial increase in latency. This suggests that there's an optimal operating range for multi-server setups where they are effective in maintaining low latency under heavier loads.

5. Scaling Limitations: Eventually, even with multiple servers, the latency begins to spike as the throughput reaches higher levels. This indicates the scaling limitations of the current setup. As the load continues to increase, the system may require additional optimizations or increased capacity to maintain low latency.

6. Inflection Point: For the 1-server setup, the inflection point (where latency begins to increase more rapidly) occurs at a much lower throughput than for the 2-server and 3-server setups, demonstrating the limitations of a single-server configuration in handling higher loads.

5 Improvements

This key-value store could be improved through multiple means, including better load balancing strategies, server tuning, and caching mechanisms.