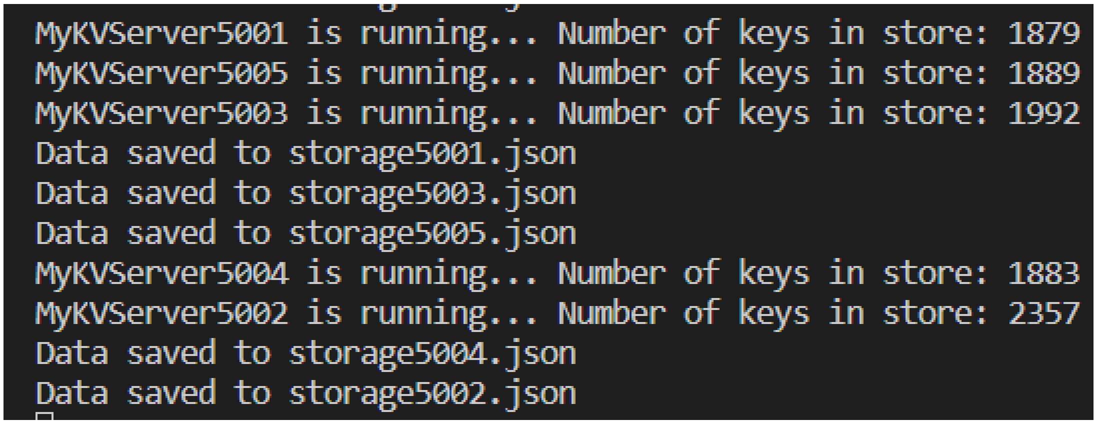
To develop a Key-Value Store with Consistent Hashing across multiple instances, I began by creating a custom server class in Python. This allowed for deployment across various processes. I re-used the Flask server from Coding Assignment 1 for the MyKVStore Class. It initializes with a server name, disk storage name, and a specified port number.

Once the key-value store server class was operational, I aimed to integrate consistent hashing to balance the request load. After considering different hashing libraries, I chose hashlib and bisect for ease of learning and system design. I created a HashRing Class that accepts a list of physical server nodes and assigns each a number of virtual nodes for distribution in the ring. For simplicity, each server node is represented by its running port rather than the instance itself. Since this application does not have security requirements, I picked MD5 from hashlib as my hashing function. The straightforward hashing method also indicated the hashing ring was going to have a 128-bit key space. Therefore, with the large key space, I set the default replication of virtual nodes as 100 to provide more balanced ID buckets for each server. To form the ring structure, I utilized a sorted list to store the keys of virtual nodes and applied the bisect library to help me maintain the sorted state of the list. For example, bisect.bisect_left() was used to find keys of an existing server node during node deletion process meanwhile the bisect.bisect_right() was used to mimic the behavior of matching a random circle position to the right nearest virtual node.

Then, I realized that I do not have a network distributor or reverse proxy server for me to incorporate the HashRing Class. Due to my limited experience and knowledge in computer networks, it was a challenge for me to implement HAProxy to my program even after spending sufficient time for test-and-trial. Therefore, I developed a low level solution to this problem. I created a MyDistrubutor Class to serve a middleware server handling client requests on port 5000 and interfacing with the HashRing. Although this provided basic functionality, the system was not fully robust in terms of some network issues.

Next, I addressed the system's scalability by enabling the addition and removal of server nodes from the hashing ring. MyDistributor could now process 'add-server' or 'remove-server' requests through POST methods. It either integrated a new node into the HashRing and initiated a server on the next sequential port, or it sent a shutdown signal to a node and redistributed tasks and data to the subsequent server in the ring. With the system running smoothly, I containerized it for deployment on Docker.

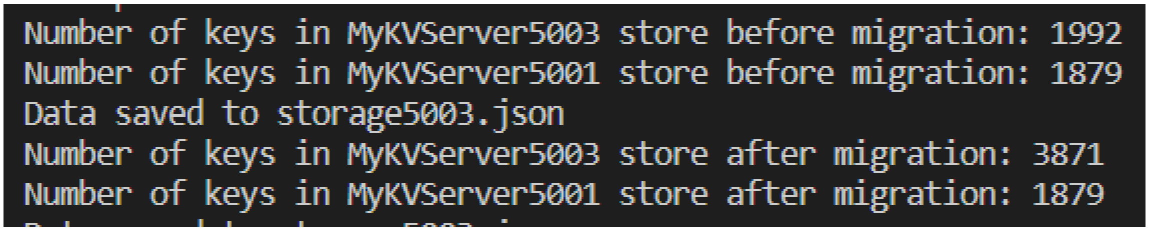
During the development, I concurrently refined myClient.py as well for testing. The script is based on example code given by Professor Annwar and adjusted to fit my server's architecture (it is also pushed to the same Github repo). It measures the latency/throughput of the system given different testing flags in the script and tests the minimum fault-tolerant functionality. As illustrated in Figure 1, the system's distribution of tasks using consistent hashing across five kv-stores is fairly even, besides MyKVServer5002 handling a larger share. Further testing and adjustments are planned to optimize the distribution.



```
MyKVServer5001 is running... Number of keys in store: 1879
MyKVServer5005 is running... Number of keys in store: 1889
MyKVServer5003 is running... Number of keys in store: 1992
Data saved to storage5001.json
Data saved to storage5003.json
Data saved to storage5005.json
MyKVServer5004 is running... Number of keys in store: 1883
MyKVServer5002 is running... Number of keys in store: 2357
Data saved to storage5004.json
Data saved to storage5002.json
```

Figure 1: A snapshot of client request distribution

In addition, the test script also demonstrates the behavior of the failure of a physical node and the ensuring redistribution. The result shown in Figure 2 illustrates the functionality with port 5001 shutting down and port 5003 being the next server (3871 = 1992 + 1879). To improve the fault tolerance, the next server instantly saves current local storage to disk before and after migration. This simulation ensures that disk storage remains unaffected by server failure, as why the number of keys in 5001 remains the same.



```
Number of keys in MyKVServer5003 store before migration: 1992
Number of keys in MyKVServer5001 store before migration: 1879
Data saved to storage5003.json
Number of keys in MyKVServer5003 store after migration: 3871
Number of keys in MyKVServer5001 store after migration: 1879
```

Figure 2: A snapshot of add/remove server test

Lastly, I assessed my system's performance under various conditions. The test script records latency and throughput on the client side, saving the results to a CSV file. I tested the default system with 5 KV stores with respect to the increase of clients and the increase of requests per client. The results are shown in Figure 3 and Figure 4 respectively (I apologize for my poor data visualization skills. I was not able to combine both latency and throughput into one line chart with different scales, so I had to use bar graphs as a substitute). It is an expected trend in the increase in latency and decrease in throughput when network load increases in both scenarios

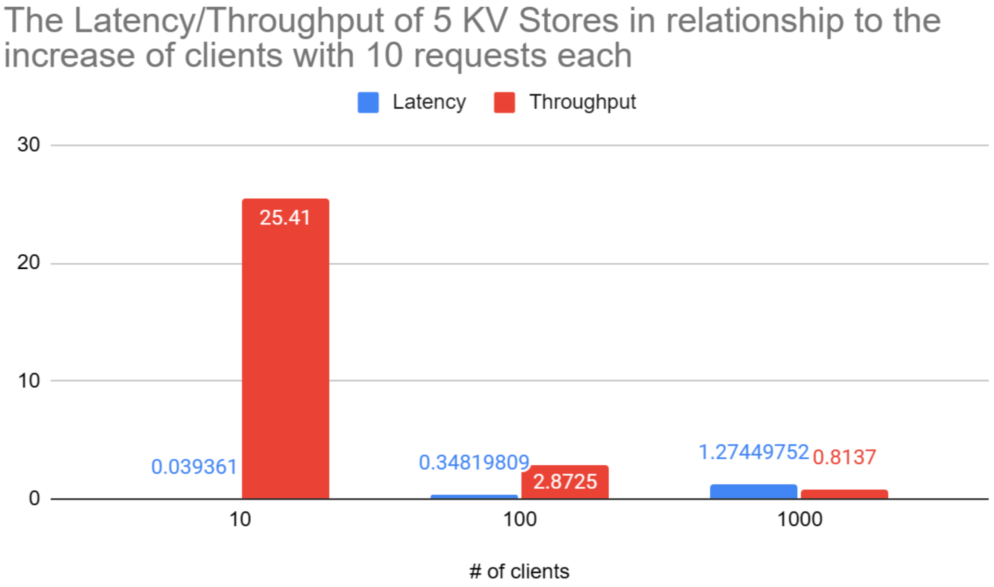


Figure 3: A bar graph showing the relationship between system performance and the increase of clients

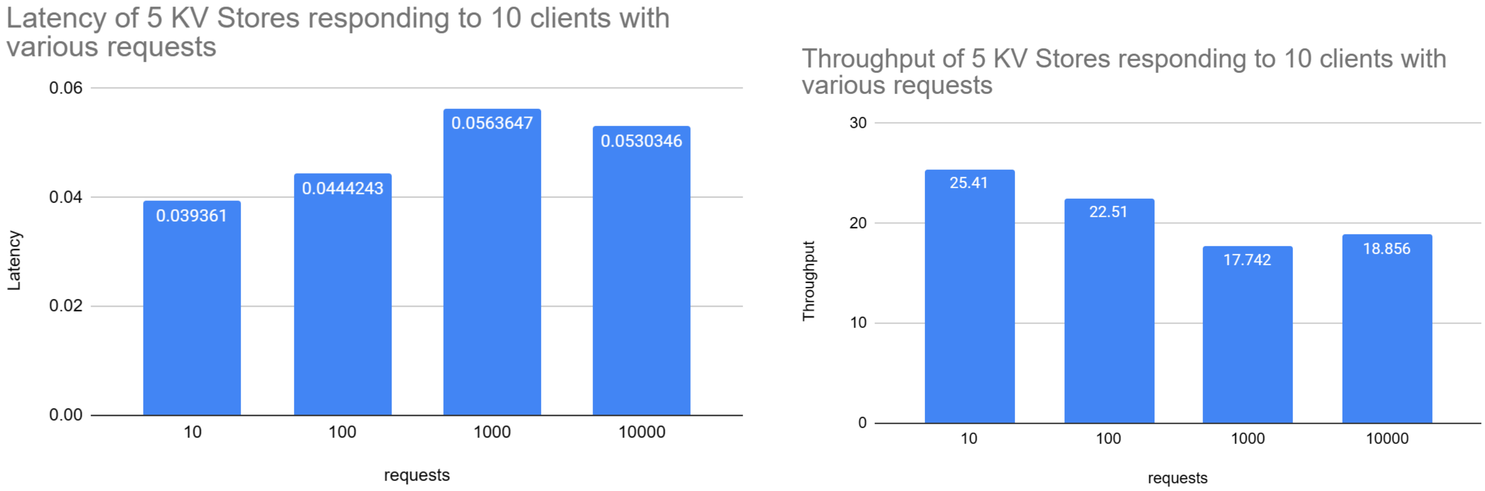


Figure 4: two bar graph showing the relationship between system performance and the increase of request per clients (10 clients in total)

I also measured the performance of my system with respect to the number of KV stores (1, 3, 5, 7, 15, and 30), while keeping all other variables constant (100 clients and 10 requests). The result is shown in Figure 3 below, which unexpectedly shows latency increases and throughput decreases as the number of KV store servers increases. This may stem from using MyDistributor instead of a professional load balancing solution, and the way servers are created and run could be affecting performance as well. I will apply optimization to the system in the future. Moreover, I also measured the performance of the system in response to different target clients, such as read main, write main, etc, the results are shown in Figure 6.

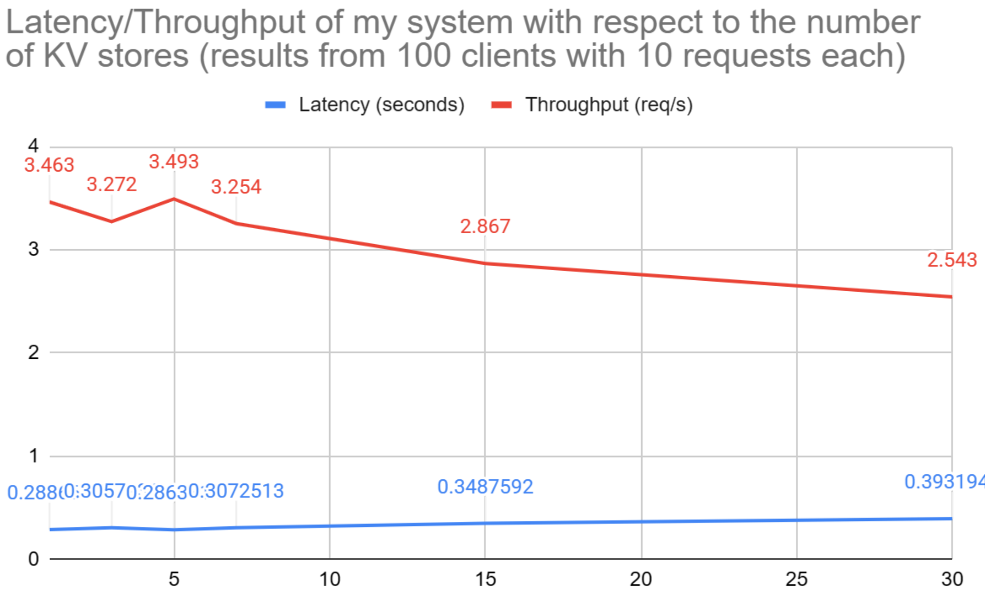


Figure 5: A graph shows the relationship between system Latency/Throughput output and the increase of KV stores

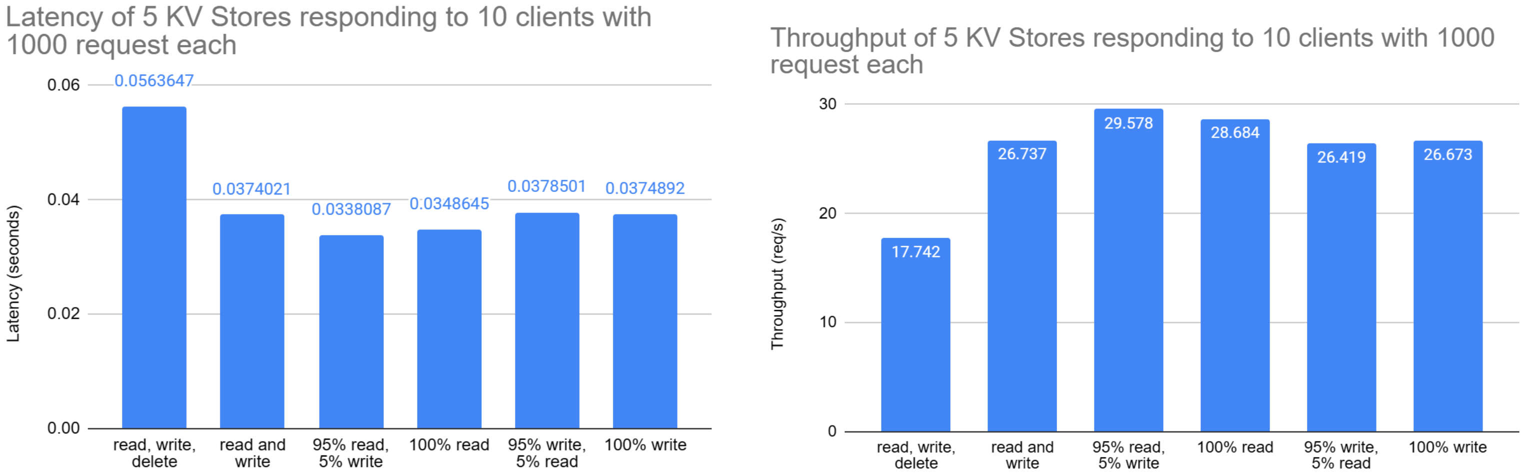


Figure 6: Two bar graphs showing how the system responses to different target clients