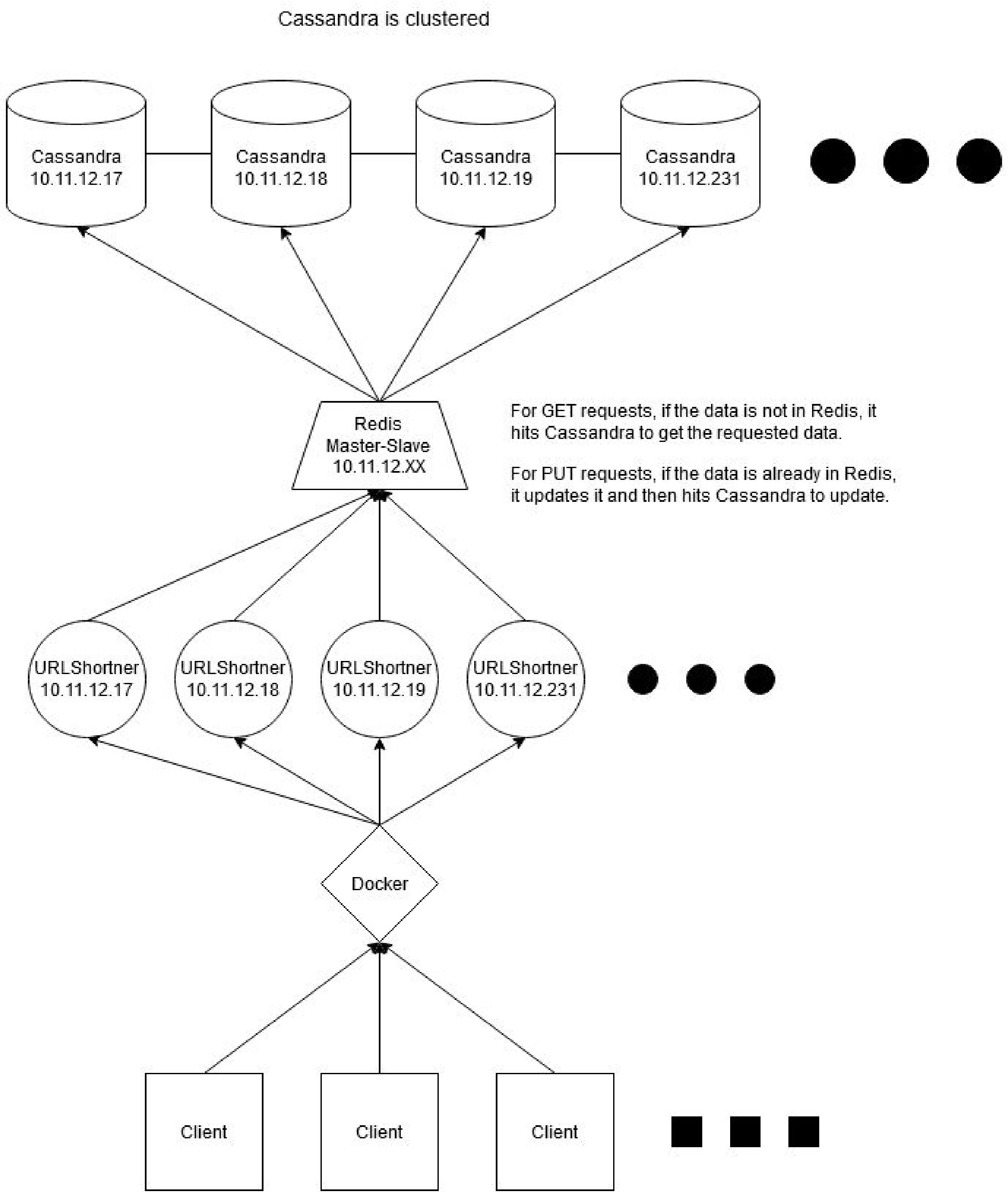
Assignment 2 - group77 - sharifp6, sahgalro, maguang

Getting Started: ./dev.sh start

System Architecture:

Below is a diagram laying out the architecture of our URL Shortener service.



Design:

For our system, we use Flask/gunicorn, Redis, Cassandra and Docker. Docker acts as the Load Balancer/Reverse Proxy. Flask/Python is the main app (URLShortner) that satisfies the requests. We use Redis as our caching system since getting data from Redis is much faster than Cassandra. Cassandra is our persistent data storage. We also use gunicorn in order to be able to process more request calls with each Flask app instance since Flask alone only accepts one request at a time so using gunicorn helps with the performance and processing more requests.

Cassandra:

For Cassandara, we are using a replication factor of 2. Since there are only 5 hosts, we felt like 2 would balance our performance between PUTs and GETs. Since GETs also use caching, the performance for GETs would not be hit too hard by lowering the number of the replications, on the other hand, it benefits PUT performance.

Redis:

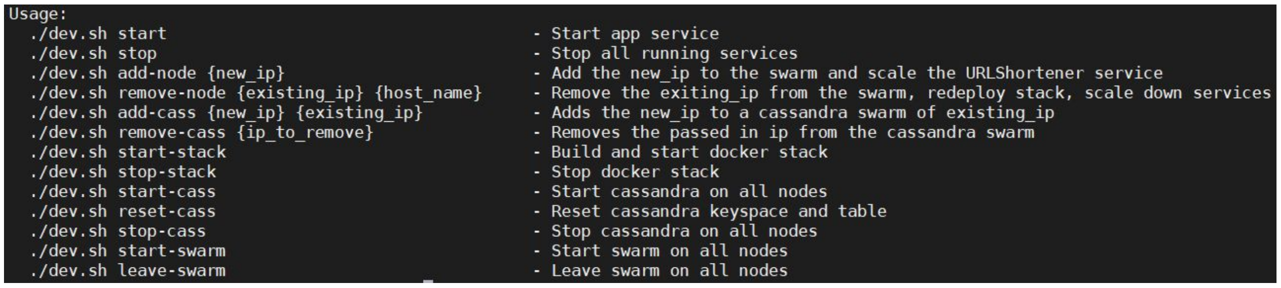
We have a master-slave Redis. We use Redis for caching GET calls. When the system receives a GET call it first checks Redis. If the data exists in Redis then it never hits Cassandra. For PUT calls, if the data exists in Redis, it gets updated and then Cassandra is called to also update the data in Cassandra. We also tried using Redis pub/sub in order to make PUTs faster by storing the received data into Redis and then using publish to notify another application in the background to update Cassandra from Redis. However, it made the performance worse, and therefore we abandoned the idea. The reason we are only using a master-slave Redis in one host is because clustering Redis in docker swarm is difficult (impossible, we did a lot of research but could not find a solution). Therefore, we decided to only use one Redis master-slave container and mainly use it as cache.

Monitoring:

For monitoring, we created a Flask app that checks every host available and looks for the available containers whenever it gets hit. It also updates every 5 seconds when open in the browser.

Orchestration:

We have a bash script that serves as the orchestration/entry point for our system. Additionally, we have a **nodes** file that contains the nodes part of the swarm on initialization file. This file gets updated when nodes are added and removed (./dev.sh add-node, ./dev.sh remove-node).
This orchestration script t is ran using **`./dev.sh`** and can be followed with the following commands:

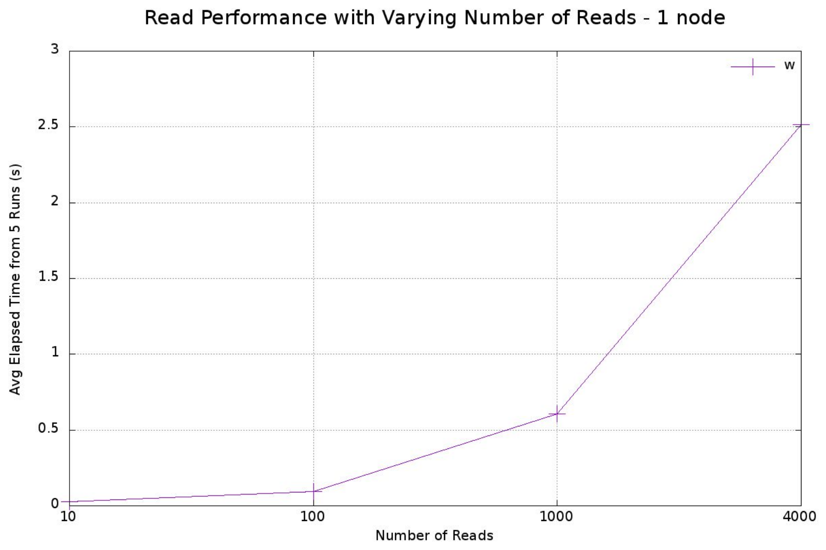


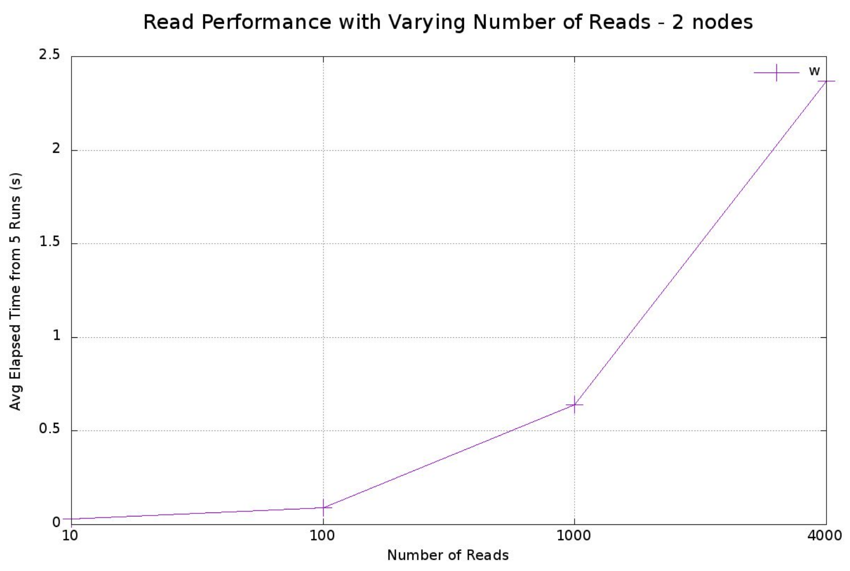
```
Usage:
  ./dev.sh start                              - Start app service
  ./dev.sh stop                               - Stop all running services
  ./dev.sh add-node {new_ip}                  - Add the new_ip to the swarm and scale the URLShortener service
  ./dev.sh remove-node {existing_ip} {host_name}     - Remove the exiting_ip from the swarm, redeploy stack, scale down services
  ./dev.sh add-cass {new_ip} {existing_ip}    - Adds the new_ip to a cassandra swarm of existing_ip
  ./dev.sh remove-cass {ip_to_remove}         - Removes the passed in ip from the cassandra swarm
  ./dev.sh start-stack                        - Build and start docker stack
  ./dev.sh stop-stack                         - Stop docker stack
  ./dev.sh start-cass                         - Start cassandra on all nodes
  ./dev.sh reset-cass                         - Reset cassandra keyspace and table
  ./dev.sh stop-cass                          - Stop cassandra on all nodes
  ./dev.sh start-swarm                        - Start swarm on all nodes
  ./dev.sh leave-swarm                        - Leave swarm on all nodes
```

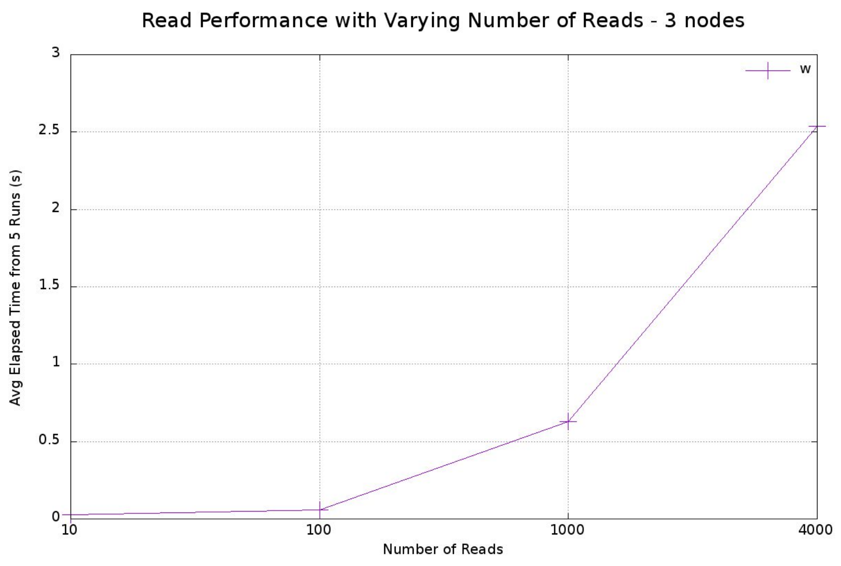
Testing:

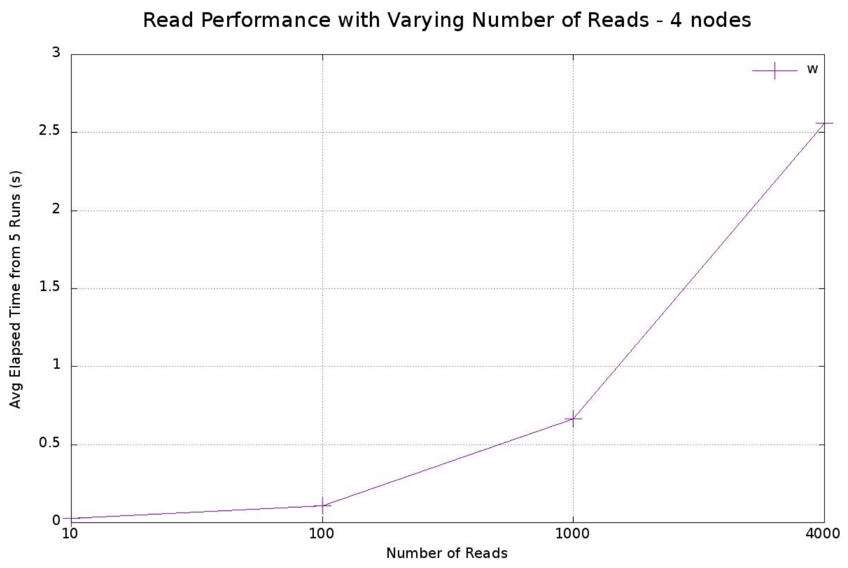
Both performance and availability were tested through scripts under `/performanceTesting` and the corresponding graphs can be shown below. For performance, both read and writes with the system were tested with a varying number of nodes, and varying reads/writes. These graphs can also be found under `/performanceTesting'. The following tests were performed with 4 threads on each gunicorn worker.

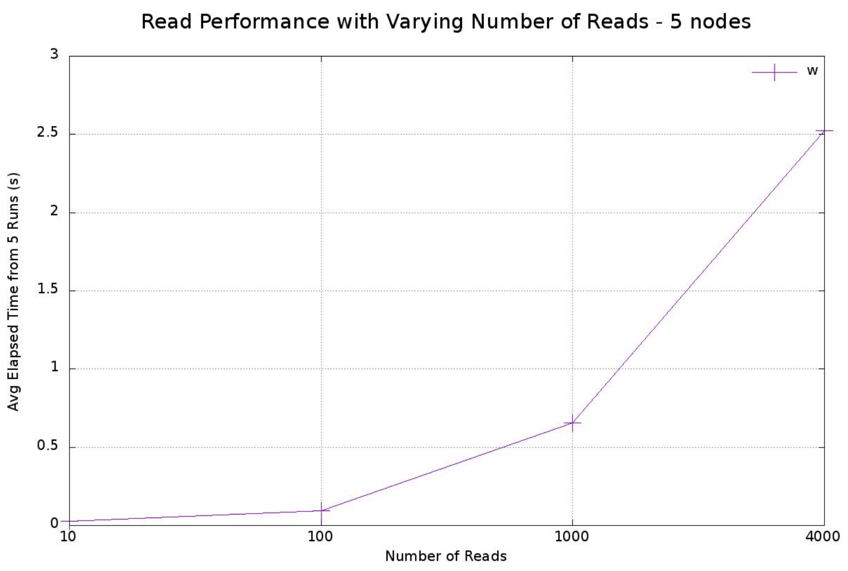
Reads




Read Performance with Varying Number of Reads - 2 nodes
w
Avg Elapsed Time from 5 Runs (s)
Number of Reads
0
0.5
1
1.5
2
2.5
10
100
1000
4000

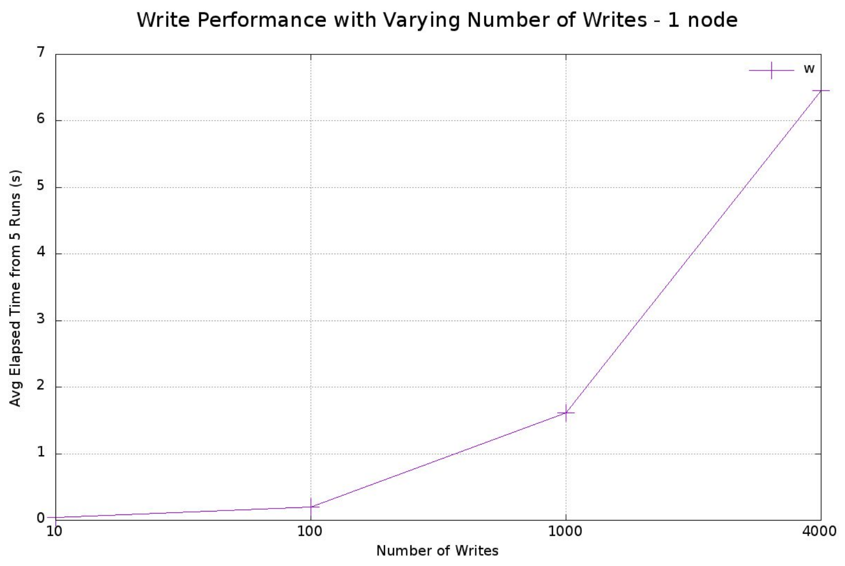

Read Performance with Varying Number of Reads - 3 nodes
w
Avg Elapsed Time from 5 Runs (s)
Number of Reads
0
0.5
1
1.5
2
2.5
3
10
100
1000
4000

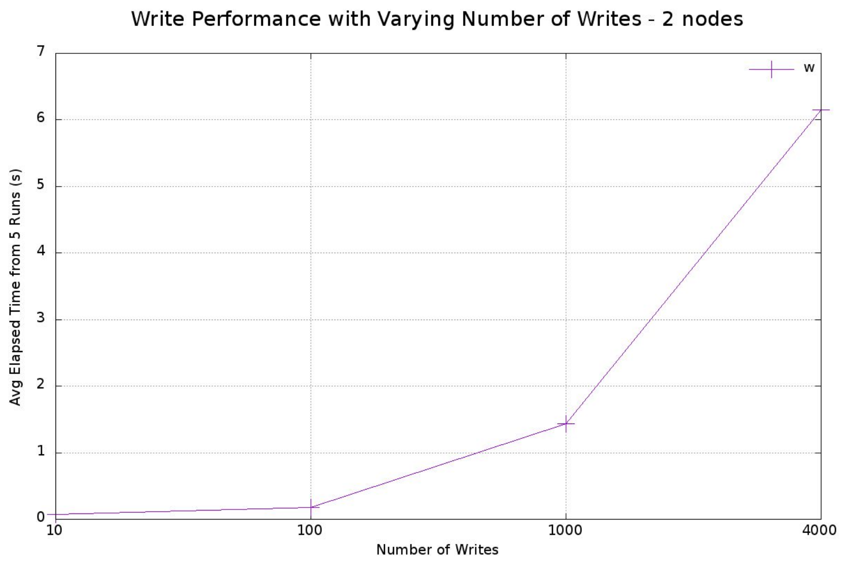

Read Performance with Varying Number of Reads - 4 nodes
w
Avg Elapsed Time from 5 Runs (s)
Number of Reads
0
0.5
1
1.5
2
2.5
3
10
100
1000
4000

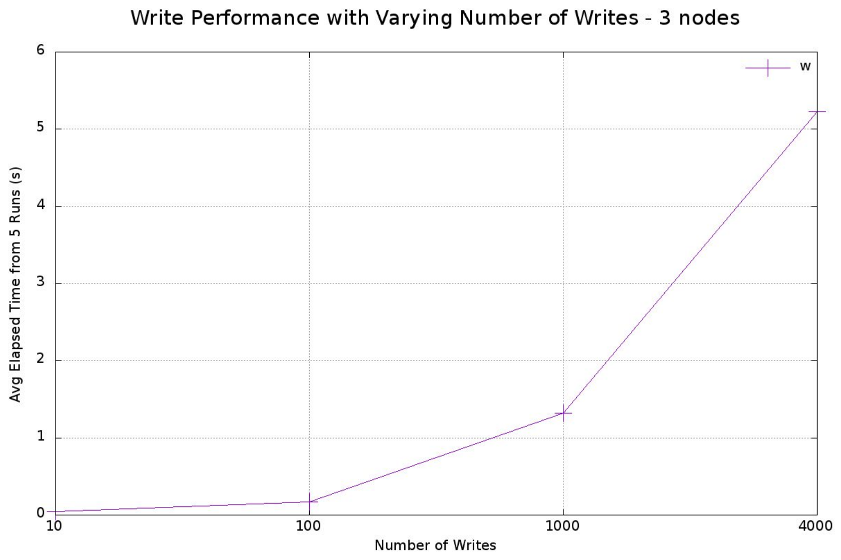


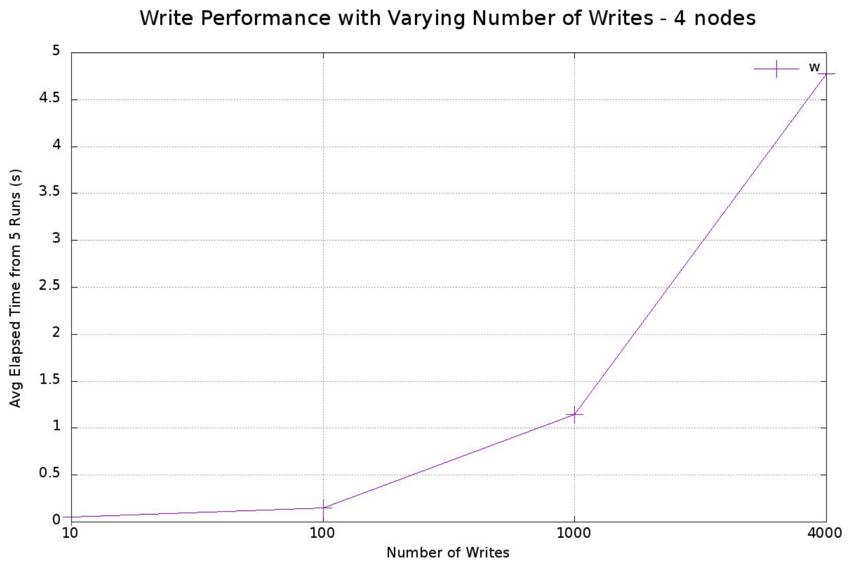
For a varying number of reads, the performance was consistent across a varying number of nodes (1-5). This was understandable as it means the GETs were always using redis as we were sending the same address each time.

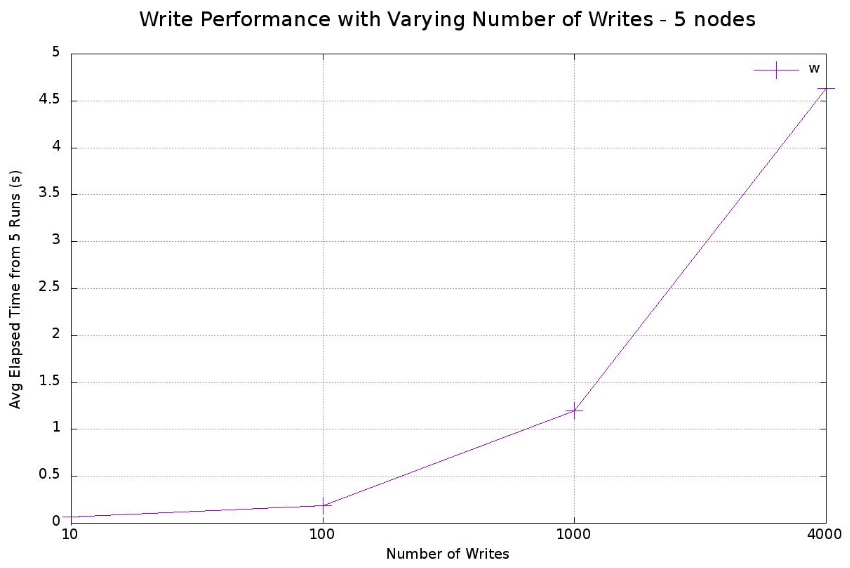
Writes




Write Performance with Varying Number of Writes - 2 nodes
Avg Elapsed Time from 5 Runs (s)
Number of Writes
w
0
1
2
3
4
5
6
7
10
100
1000
4000


Write Performance with Varying Number of Writes - 3 nodes
Avg Elapsed Time from 5 Runs (s)
Number of Writes
w
0
1
2
3
4
5
6
10
100
1000
4000


Write Performance with Varying Number of Writes - 4 nodes
Avg Elapsed Time from 5 Runs (s)
Number of Writes
w
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
10
100
1000
4000



For writes, the performance gain on a varying number of nodes was evident as speeds went from 6.5s (1 node) to ~4.5s (5 nodes). A 40% improvement in speed.

Strengths:

- Good partitioning and replication
- Strong orchestration for setting up the system and adding and removing hosts.
- Good performance when using apache bench to test our system

Weaknesses:

- Caching can be improved
- After ~12-14 thousand subsequent requests Flask freezes for a long time. We investigated this issue and realized the problem is with Flask. Even empty requests have the same issue.