## **1. System Architecture**

* **Application System**: Flask app, Redis, and Cassandra.
* **Monitoring System**: Docker health checks monitor service status and logging
* **Data Flow**: Requests go through Flask, with reads and writes handled by Redis (cache) and Cassandra (persistent storage). If a write request, we first write to Cassandra, and then to redis-master which publishes to redis-slaves. If a read request, we try reading from Redis cache if data is available, if not we read from Cassandra and add to redis-master which also publishes to redis-slaves. Redis temporarily stores data when Cassandra is down, and if Redis is down, cache is simply bypassed and Cassandra is used without caching.

### **2. System Components and Guarantees**

1. **Consistency**: Cassandra (QUORUM writes) and Redis cache; Redis pub/sub, master/slave architecture is used to ensure consistency. If there is no Cassandra connection available, data is stored in-memory on flask until Cassandra is available. If there is no Redis available, data is stored/retrieved from Cassandra.
2. **Availability**: Docker’s restart\_policy attempts to automatically restart python-app, redis-master, redis-slave, and visualizer services upon failure of health checks, maximizing uptime. Failure is detected through health checks of docker and the health check endpoints on our flask apps. If there is no Cassandra connection available, data is stored in-memory on flask until Cassandra is available.
3. **Partition Tolerance**: Redis master-slave and Cassandra replication allow operations to continue during network partitions. Docker stack also will utilize the available nodes to redeploy one of the services as a container if a service node goes down
4. **Data Partitioning**: Cassandra’s hash-based partitioning distributes data evenly across nodes. Redis as cache has a master-write node and a slave-read node where readers sub to master that pubs.
5. **Data Replication**: Cassandra replication (factor of 2) and Redis master-slave ensure data durability. There are 2 Redis slaves to one master. We ensure that the master and slaves are different nodes.
6. **Load Balancing**: Redis caches frequently accessed data; Cassandra’s hash partitioning distributes data. Docker will pick one of the Redis slave nodes and use it, and will fail over to the other. Since Redis slave nodes are subbed to Redis master, this does not reduce consistency. Adding more nodes and more flask URLShortener replicas will allow the requests to be distributed across the replicas.
7. **Caching**: Redis stores frequently accessed data. Redis-master is for writing. It publishes to 2 Redis-slaves, which are for reading.
8. **Process Disaster Recovery**: Docker restart policies automatically recover failed services. Health check functionality is utilized to check whether a service has gone down.
9. **Data Disaster Recovery**: Redis persists to disk (using volumes feature of docker), and Cassandra replication provides longer-term durability since data is hashed across nodes.
10. **Orchestration**: We use a mixture of bash scripts and docker swarm’s API to Orchestrate our cluster.
    1. The bash scripts do the following:
       1. start/stop Cassandra cluster
       2. add new node to Cassandra cluster
       3. Remove a Cassandra node (turning an instance of Cassandra off)
       4. Start the swarm cluster
       5. shutdown the swarm cluster
       6. Adding a new node to swarm
       7. Deleting a node from the swarm (this is done through a combination of docker swarm commands and scripts)
11. **Health Check**: Docker health checks and the health check endpoints of our app ensure services are functional and trigger restarts if needed.
12. **Horizontal Scalability**: Adding more Cassandra nodes and Redis replicas increases failover nodes, but not necessarily computational power. Storage will increase when we run a new instance of Cassandra on a new node. Adding more replicas of the flask URLShortener application across more nodes will allow for increase in computational power.
13. **Vertical Scalability**: Cassandra will have more storage to utilize, and it can also benefit from faster secondary storage devices (SSDs over mechanical disks). Faster and more RAM would increase the performance of the services run on the swarm cluster, especially Redis and our flask application. CPU compute would also just allow speed up of processes.

## **3. System Performance**

### **Load Test Results**

**LoadTest1, 3 nodes, 3 python-apps, 3 cassandra nodes:**

**Server Software: Werkzeug/3.0.6**

**Server Hostname: localhost**

**Server Port: 4000**

**Document Path: /000000000000000000000000000000000000000**

**Document Length: 26 bytes**

**Concurrency Level: 10**

**Time taken for tests: 18.574 seconds**

**Complete requests: 10000**

**Failed requests: 0**

**Non-2xx responses: 10000**

**Total transferred: 1980000 bytes**

**HTML transferred: 260000 bytes**

**Requests per second: 538.40 [#/sec] (mean)**

**Time per request: 18.574 [ms] (mean)**

**Time per request: 1.857 [ms] (mean, across all concurrent requests)**

**Transfer rate: 104.10 [Kbytes/sec] received**

**Connection Times (ms)**

**min mean[+/-sd] median max**

**Connect: 0 1 23.2 0 1059**

**Processing: 12 18 5.7 16 81**

**Waiting: 2 7 5.1 5 65**

**Total: 12 19 24.0 16 1085**

**Percentage of the requests served within a certain time (ms)**

**50% 16**

**66% 18**

**75% 20**

**80% 21**

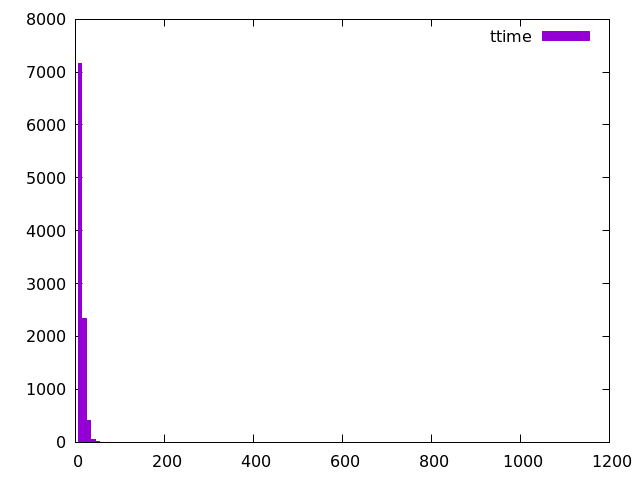
**90% 25**

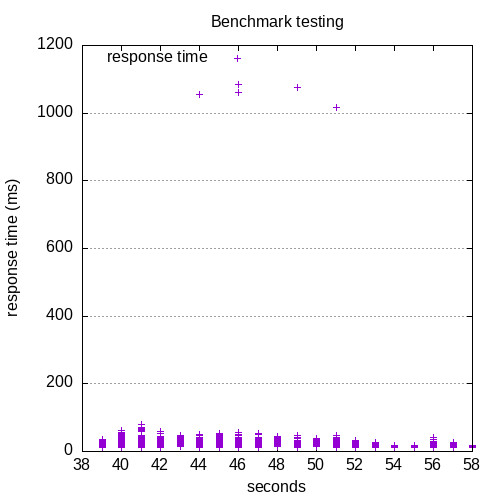
**95% 29**

**98% 34**

**99% 38**

**100% 1085 (longest request)**

****

****

**LoadTest2, 3 nodes, 3 python-apps, 3 cassandra nodes**

**Server Software: Werkzeug/3.0.6**

**Server Hostname: localhost**

**Server Port: 4000**

**Document Path: /000000000000000000000000000000000000000**

**Document Length: 26 bytes**

**Concurrency Level: 10**

**Time taken for tests: 17.800 seconds**

**Complete requests: 10000**

**Failed requests: 0**

**Non-2xx responses: 10000**

**Total transferred: 1980000 bytes**

**HTML transferred: 260000 bytes**

**Requests per second: 561.80 [#/sec] (mean)**

**Time per request: 17.800 [ms] (mean)**

**Time per request: 1.780 [ms] (mean, across all concurrent requests)**

**Transfer rate: 108.63 [Kbytes/sec] received**

**Connection Times (ms)**

**min mean[+/-sd] median max**

**Connect: 0 0 20.7 0 1055**

**Processing: 12 17 4.6 16 57**

**Waiting: 2 6 4.1 5 44**

**Total: 12 18 21.2 16 1069**

**Percentage of the requests served within a certain time (ms)**

**50% 16**

**66% 18**

**75% 19**

**80% 20**

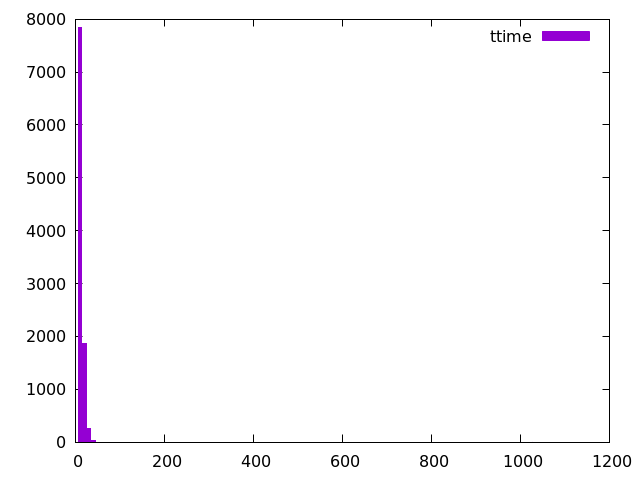
**90% 23**

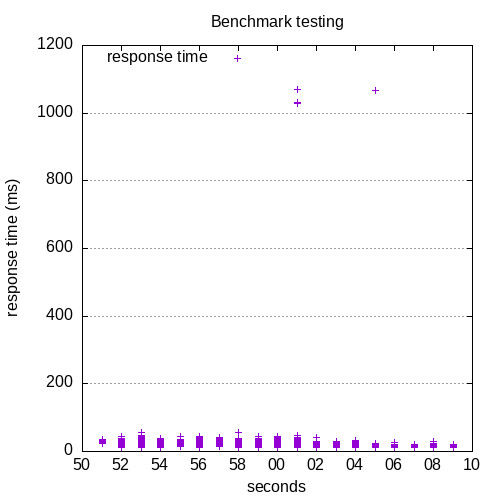
**95% 27**

**98% 32**

**99% 35**

**100% 1069 (longest request)**

****

****

## **5. Testing Tools**

1. **ab (Apache Benchmark):** measures the performance of a system on a significant quantity of requests
2. **LoadTest.java**: Generates PUT and GET requests to simulate various load conditions, measuring stability and performance.
3. **curl and Docker Logs**: Verifies endpoint functionality and monitors health checks and container restarts.

## **6. Comparison with A1**

Comparing this data with the A1 response times we see that the response times are significantly higher:

The median time for LoadTest1 is 16ms for A2 while it is 11.5ms for A1. It's also important to note that the highest response time for LoadTest1 is 1000ms+ for A1 while <100ms for A2. Although most of this latency comes from the connection portion of the loadtest.   
  
The median time for LoadTest2 is also 16ms for A2 while it is 9ms for A1. We see the same pattern of long connection time for A2 on the Loadtest with the max latency being 1000ms+.

The increase in latency is, however, expected. We are using a tech stack that uses python which is not as performant as Java used on A1.