CSC409 Group88 Assignment 2 Report

Jingwen (Steven) Shi

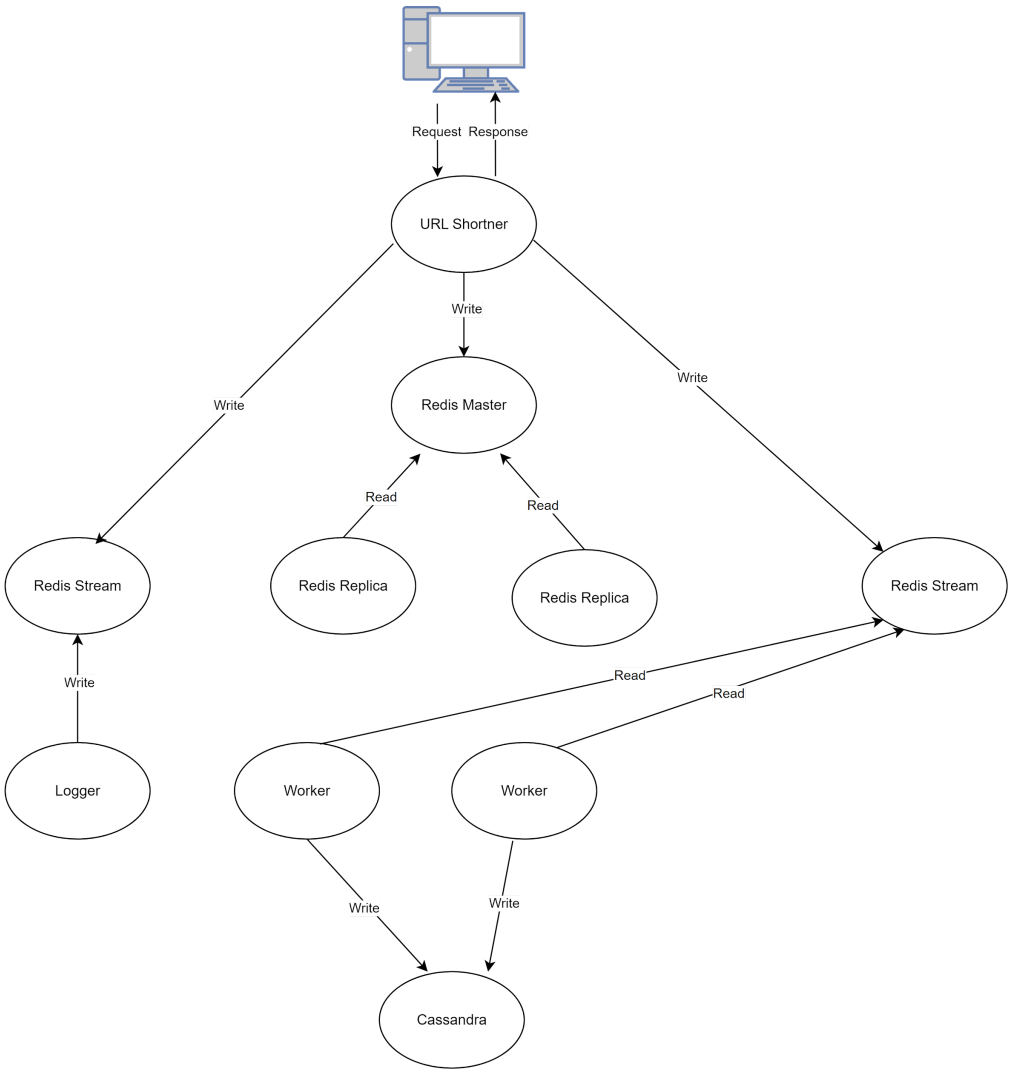
Feiran (Philip) Huang

Xuankui Zhu

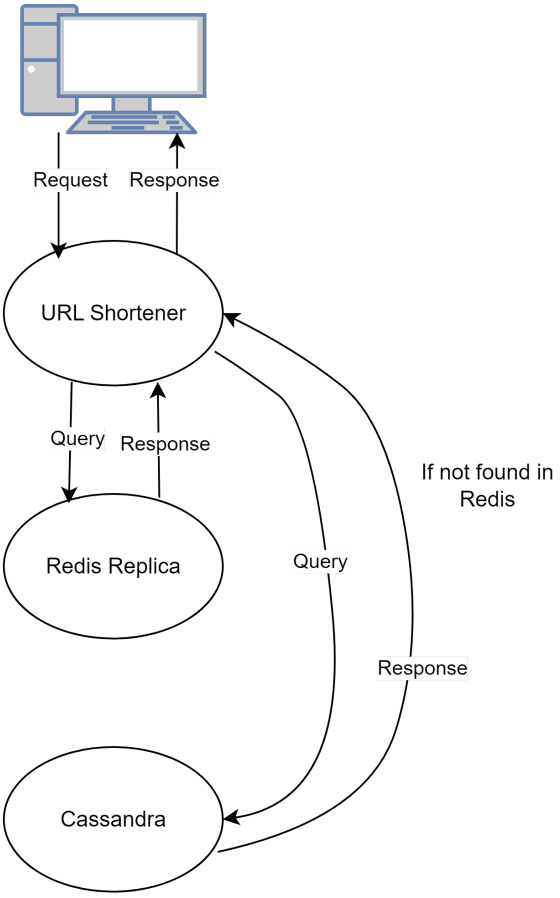
All services are run in Docker containers on separate virtual machines within a virtual network.

A Docker swarm is used to manage the virtual machines, or nodes within the cluster. The swarm has a single manager node that does load balancing by distributing work to all nodes within the cluster.

System Diagrams



PUT diagram


Request
Response
URL Shortener
Query
Response
Redis Replica
Query
If not found in
Redis
Response
Cassandra

GET diagram

List of Services

Redis Primary

Redis has one master server which all replica servers duplicate from. When the web application running on a node receives a PUT request, it saves the request data into the Redis master server. All replica Redis servers running on all nodes will asynchronously update their data with respect to the master server. The master server only runs on the manager node within the swarm. This makes it simpler and easier to manage, since the manager of the Docker swarm and the Redis master both exist on a single node. When the data reaches the limit, the older ones will be evicted under the policy of LRU to ensure that Redis doesn't break under extreme load.

Redis Replica

There are two replica Redis servers running in the system, which will be asynchronously updated to keep up with the master server. These replicas strictly run on non-manager worker nodes. This is done to lower the load on the manager node since it will be running the master server which handles the write requests only. When a GET request is sent to any node, the node will try to retrieve data from one of the replicas. If the data is not found within the replica, it will then retrieve the requested data from the Cassandra database. The data in all Redis instances (master and replicas) are persistent since a host directory is mounted into the containers they are running on.

URL Shortener/Handler

Instances of the web application are run on different nodes. Upon receiving a PUT request, the application stores the request in the master Redis server, while also pushing the request data into two Redis streams, which are read by all consumers of them. All worker instances are consumers of one of the streams, while the other stream is read solely by the logging service running on the manager node. After successfully pushing the request data into the streams, the web application sends a confirmation of the update back to the user. When receiving a GET request, the application tries to retrieve the requested data from the Redis replica running on the same node. If it can’t find the data, it will search for it in the Cassandra database.

Worker

Instances of workers implemented in Python are also run on different nodes. All workers are consumers of one of the Redis streams. They read PUT request data from the stream and insert them into the Cassandra database.

Since there are many workers in the system, and their reads may overlap with each other, we used the `'xack'` command to allow Redis to automatically avoid data races and each worker to remove stream entries once they are read. This way, all workers can read different request data.

Logger

A logging service runs on the manager node. It reads data from the other Redis stream not used by the workers, and records the PUT request information such as request time, short and long URLs into the manager node's hard disk.

Cassandra

A Cassandra cluster is also run in Docker but outside the swarm as required, with each Cassandra node running in a VM node's Docker container, thereby replicating and partitioning data across multiple VM nodes. Writers and URL shorteners will be retrieving data from the Cassandra node running in the same VM as they are.

Monitor

The monitoring system is served by a Python Flask backend with a webpage frontend. The backend utilizes the IP addresses of various Docker servers to establish SSH connections to the respective servers and collect information about each Docker node.

Information is served by the backend through the `'/get_docker_nodes'` endpoint. The obtained data is then processed and structured into a JSON format, encompassing details such as the node name, Docker container information, and a status message indicating success or a timeout if the SSH command takes an extended duration.

The front end accesses this endpoint at regular intervals of 5 seconds. This enables it to monitor multiple Docker nodes from a centralized location, providing timely updates on the status of the Docker servers.

Availability & Fault Tolerance

The system guarantees availability for PUT requests by having the front-end service respond to all PUT requests instantly. For GET requests, the system is also always able to respond by either retrieving data from Redis or Cassandra. Redis instances are always able to restart when failures are detected, while Cassandra has its own availability guarantees.

Docker swarm will automatically load balance the request and send it to the services that are online. It will then automatically try to restart the service on failure to ensure fault tolerance.

Scalability

The docker swarm will automatically handle the case of adding a new node to the swarm. Then the service can be scaled up or down with respect to the number of replicas you update. And data scalability is handled by Cassandra with virtual nodes.

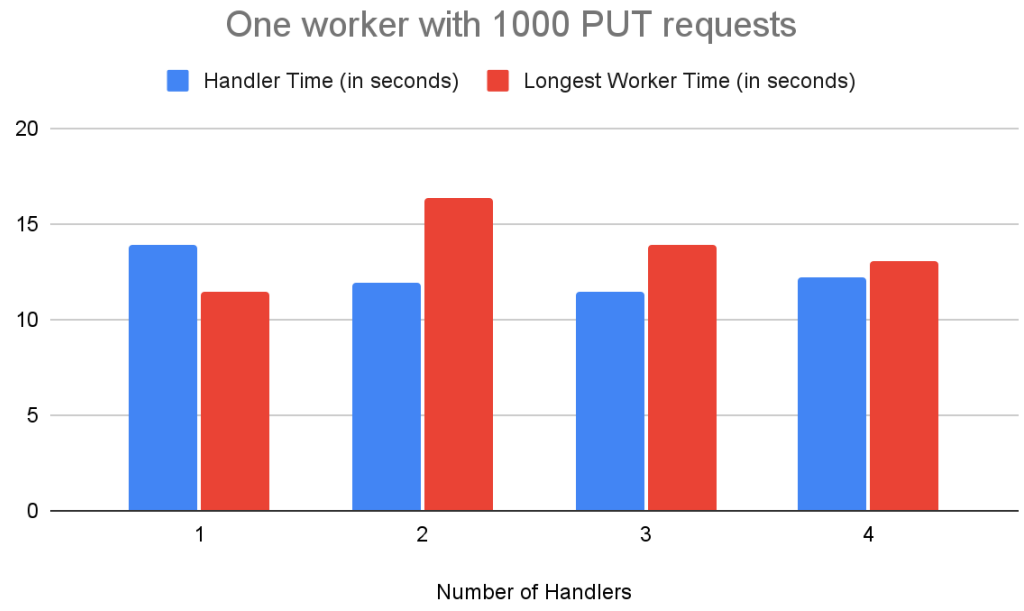
Consistency

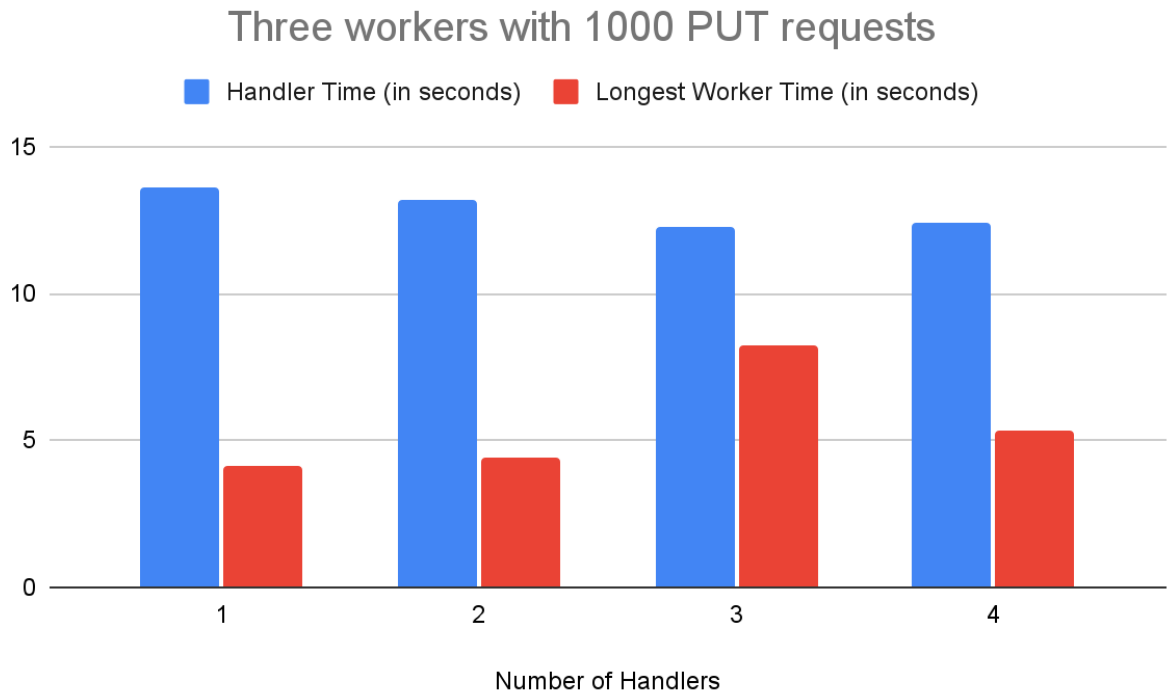
The system is not 100% consistent since Redis secondaries updates/backups from the primary asynchronously. Also, it takes time for the worker to read the data from Redis Steam and then write to Cassandra. Under extreme cases, the user may send a GET request immediately after sending a PUT request to our service. In this case, the Redis secondaries may not be able to sync the data with the primary and the worker has not written to Cassandra yet, which results in a 404 error code.

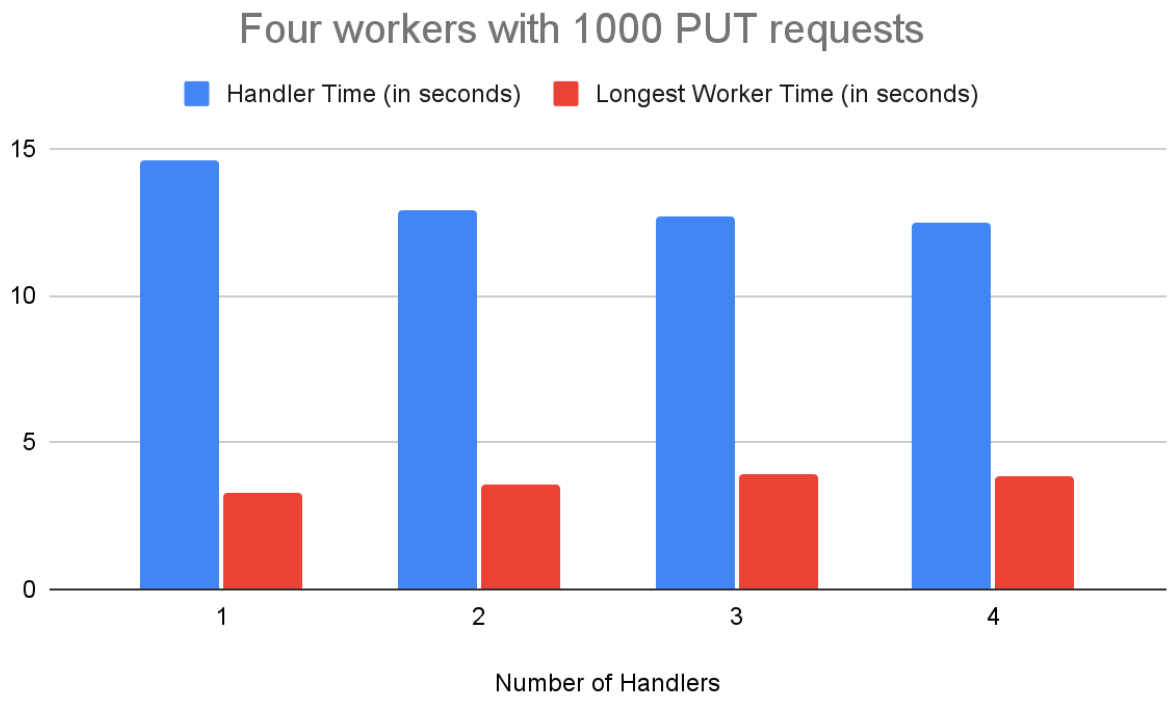
Performance Testing

Since the system contains two parts, URL handler and worker/writer, we recorded the run time of each part separately.

We tested the performance URL handler with LoadTest.java of 4000 PUT requests from 4 users and recorded the corresponding worst performance of the worker to get a sense of the worst-case performance of the overall system.




Three workers with 1000 PUT requests
Handler Time (in seconds)
Longest Worker Time (in seconds)
15
10
5
0
1
2
3
4
Number of Handlers


Four workers with 1000 PUT requests
Handler Time (in seconds)
Longest Worker Time (in seconds)
15
10
5
0
1
2
3
4
Number of Handlers

Number of Workers	Number of Handlers	URL Handler Processing Time (in seconds)	Longest Worker Time (in seconds)
1	1	13.934	11.4589
1	2	11.926	16.3756
1	3	11.466	13.8554
1	4	12.206	13.0739
2	1	13.162	6.6779
2	2	12.353	7.4131
2	3	12.513	7.5197
2	4	12.182	7.6994
3	1	13.61	4.1304
3	2	13.169	4.4206
3	3	12.283	8.2244
3	4	12.372	5.3269
4	1	14.597	3.3192
4	2	12.89	3.5478
4	3	12.653	3.9195
4	4	12.498	3.8743

The graphs and table above show four different tests on PUT requests conducted with different numbers of workers (Python programs that read from a Redis queue/stream and write to Cassandra) instances in the cluster. As the number of workers in the cluster increases, the longest working time (time for workers to read from Redis steam and send the request to Cassandra) drastically decreases. On the other hand, the running time of the URL-shortening web service seems to stay relatively constant, regardless of the number of URL handlers or the number of workers. This could result from the HTTP and TCP network bottleneck within the web service.

The tests show that the system is highly scalable. As the number of worker nodes increases, the total time taken by workers to update the Cassandra database is significantly decreased. Leaving HTTP as the only bottleneck, which stays relatively constant.