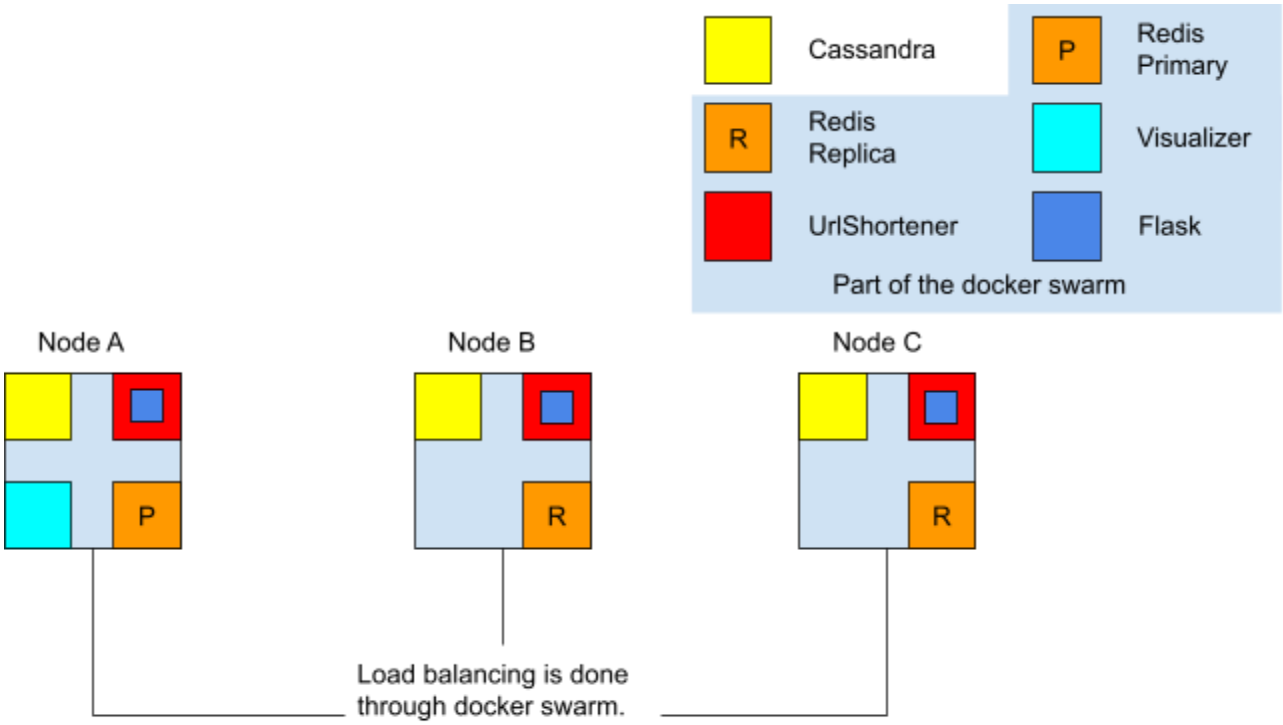
CSC409 A2 REPORT

Adrian Lam & Alan Zhao

Diagram:



Introduction:

This report details the software architecture of a Docker Swarm deployment for the url-shortener service. The Docker swarm consists of one manager node and two worker nodes. The manager node hosts the primary redis database and the docker visualizer, whereas the worker nodes host the url shortener service and the secondary redis databases.

Services:

Redis

Redis is set up with a max memory of 200mb with a LRU policy for caching. The primary redis instance on the manager nodes is writable and is persistent while the replicas on the worker nodes are read only. Since docker is stateless, the redis data is externally mounted to the

cluster vm disk for persistent data. When the url-shortner receives a write request, it will write to the primary redis if the write to cassandra is successful and when it receives a read request, it will read from the replicas and cache the response if it is not found in the cache. Our caching performance is improved when adding hosts as that will increase the number of redis replicas and give our url-shortners more redis instances to read from.

Python-Url-Shortener

This is the core service of our application. This service runs a flask web server on port 5000 that accepts GET and PUT requests. The url shortener service establishes a connection with the cassandra cluster and the redis database.

When the service receives a GET request for a short url, we first attempt to query the redis database. If we succeed resulting in a hit then we return the long url. If we fail resulting in a miss then we query the cassandra database, returning a long url if it exists.

When the service receives a PUT request with a short url and a long url, we update the cassandra database and update the redis database.

The service also logs the requests in logs/log.txt and is mounted onto the cluster vm.

Cassandra

We use a 3 node cluster setup, with a replication factor of 2. These cassandra nodes are set up outside the docker swarm. Similar to redis, the cassandra data is externally mounted to the cluster vm disk for persistent data. We have scripts to start and stop the cassandra clusters given the IP addresses as well as scripts to add and remove cassandra hosts given the IP address. The keys are repartitioned automatically when a node is added or removed. When we add a cassandra node, performance is increased as there are more nodes to process reads and writes.

Visualizer

We use the standard docker visualizer for monitoring, with custom healthchecks set up for redis and url-shortner. For redis, we use the command `redis-cli ping` to check if the redis containers are running and for the url-shortner, we use `curl --fail http://localhost:5000 || exit 1`.

Network:

We created a network called a2net which is the network of the Docker swarm. The swarm consists of a Manager node and worker nodes. The manager node holds the primary redis database and the docker visualizer.

Manager Node

The manager node hosts the primary redis database and the docker visualizer.

- The primary redis database is accessible under port 6379.
- The docker visualizer can be accessed through port 8080.

Worker Node

Python Url Shortener Service: This is the core service responsible for shortening URLs. It is connected to the following services:

- Redis_Master database for writes.
- Redis_Replica database for reads.
- Cassandra

The url-shortener service runs on port 5000 in the docker container and is mapped to 4005 on the cluster vm through docker-compose. The ./startAll.sh script stores the docker swarm token in a text file which is then used by ./addDockerNode.sh to add a new worker node to the swarm. There is also a ./removeDockerNode.sh to remove a node from the swarm.

Other:

Consistency

- Cassandra: For this application we use a cassandra cluster with an initial count of 3 nodes with a replication factor of 2. This means that each piece of data is replicated to two nodes. This improves fault tolerance and ensures that if one node goes down, the data is still available on another node.

- Redis: For this application we use Redis for caching, improving read performance. Read consistency is achieved by reading from read-only replicas of the primary Redis database on worker nodes, while overall consistency is maintained by the cassandra database.

Availability

- Docker swarm: The swarm provides availability by distributing the containers across multiple nodes, ensuring that service remains available even if one node goes down. If a node fails, Docker swarm redistributes requests to other nodes that are healthy and running.
- Redis: By having multiple replicas for reading, if one replica goes down, the url shortener service can still read from other replicas.

Data partitioning and replication

- Cassandra: The data is split among cassandra nodes, with a replication factor of 2. This means that all data can be found as long as there are two cassandra nodes.
- Redis: The data is held by Redis_Master, and replicated on Redis_replica.

Load Balancing:

- Python-Url-Shortener: The requests are equally distributed to the python-url-shortener services running inside of the swarm. The load balancing is done automatically by the swarm manager.

Healthcheck:

- The docker visualizer is running on port:8080, and has a UI to display current information of all the nodes in the swarm.
- The cassandra nodes are monitored using a bash script.

Process Disaster Recovery:

For all the services, this is handled by the docker swarm manager. It will attempt to restart containers in which processes are down.

For the cassandra nodes, the containers will also be restarted as long as the cassandra monitor is up and running.

Data Disaster Recovery:

Redis and Cassandra take care of this issue internally.

- Redis has replicas of the master database, and can promote a replica to master if the master goes down.
- Cassandra nodes automatically replicate data, and can continue functioning in the event some nodes go down.

Partition tolerance:

Redis and Cassandra take care of this issue internally.

- Redis has replicas of the master database, and can promote a replica to master if the master goes down.
- Cassandra nodes automatically replicate data, and can continue functioning in the event some nodes go down.

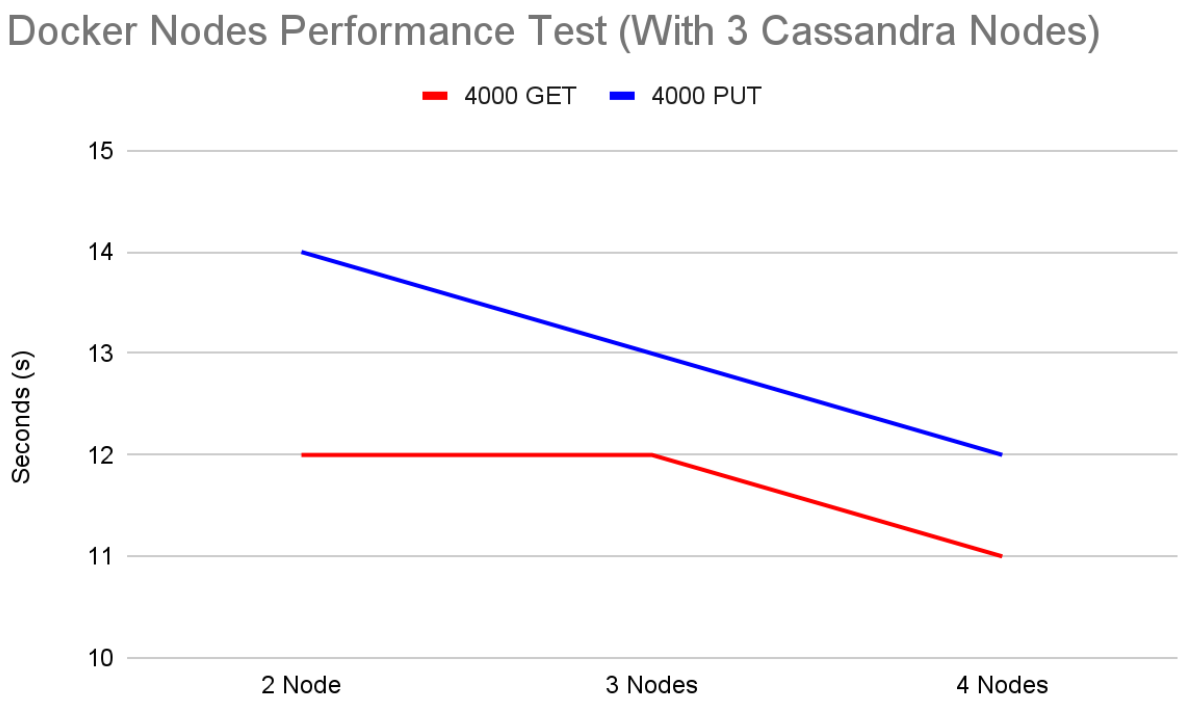
Horizontal scalability:

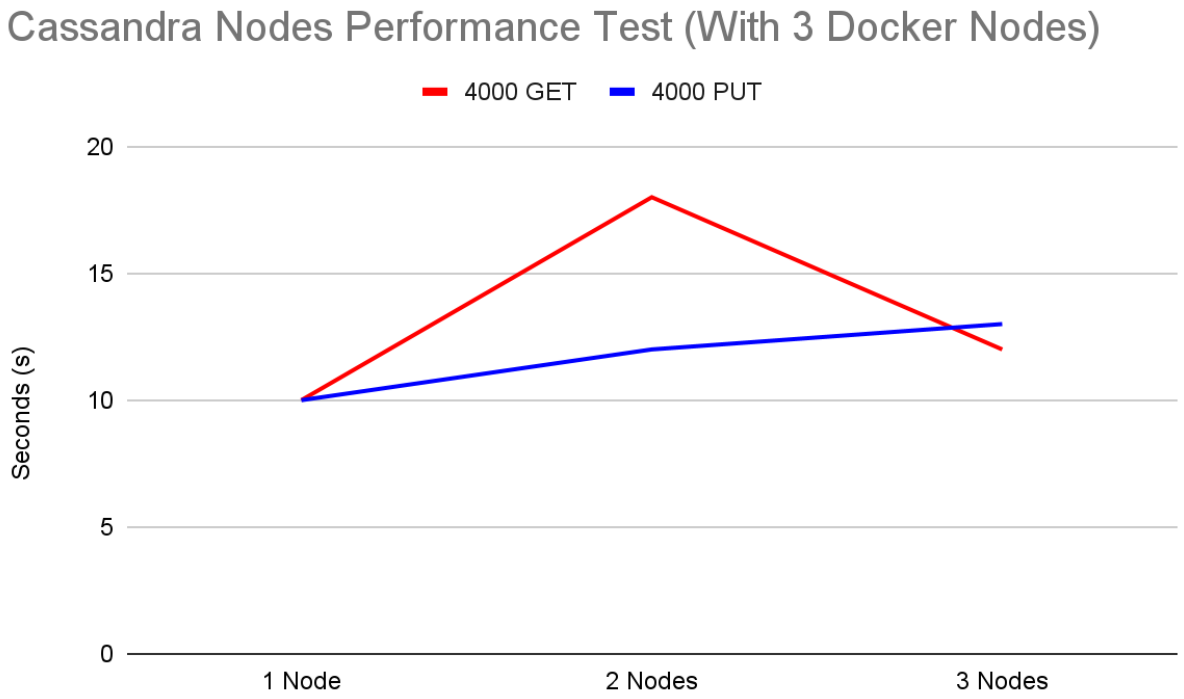
Each service increases capability when scaled horizontally.

- Python-url-shortener: We can horizontally scale this service to increase the number of requests that we can process in the application.
- Redis: We can horizontally scale this service to increase the number of replicas, and thus the read requests can be serviced quicker cache in the application.
- Cassandra: We can horizontally scale this service to increase the number of nodes in the cluster, and thus the read and write requests can be serviced quicker.

Performance Test:

Each docker node has one urlshortner and one redis instance
Testing was done using the scripts provided by Arnold, LoadTest1.bash and LoadTest2.bash.





It appears that we are getting slight performance increases for adding in new worker nodes to the swarm (a worker node includes one replica of redis and one instance of python-url-shortener). This is evidenced by our graph titled “Docker Nodes Performance Test”.

It appears that increasing the number of cassandra nodes from one to two actually worsens our performance.

Weaknesses:

We did not manage to implement version 2, with the queue and writer program. In this version, the application is bottlenecked by Cassandra. This explains why we did not see an improvement when changing the number of Cassandra nodes.

We use docker swarm for loadbalancing instead of an external loadbalancer such as nginx. This configuration is not as flexible.

We couldn’t figure out how to display the cassandra node statuses on the docker visualizer so we monitor and restart the cassandra nodes using a separate script.

In the python-url-shortener service, we used Flask. Flask (by itself) is not very powerful for handling large amounts of requests, as it is single-threaded and can only handle one request at a time. To workaround this, we tried to use waitress (a production-quality pure-Python WSGI server) as an entrypoint instead, but it made our service slower. The waitress server should create 4 worker nodes running an instance of our flask app and delegate any requests to the 4

worker nodes. However, since it made our application slower, we decided to just use Flask (by itself). It is possible that we tried other alternatives such as Gunicorn, however we ended up running out of time.