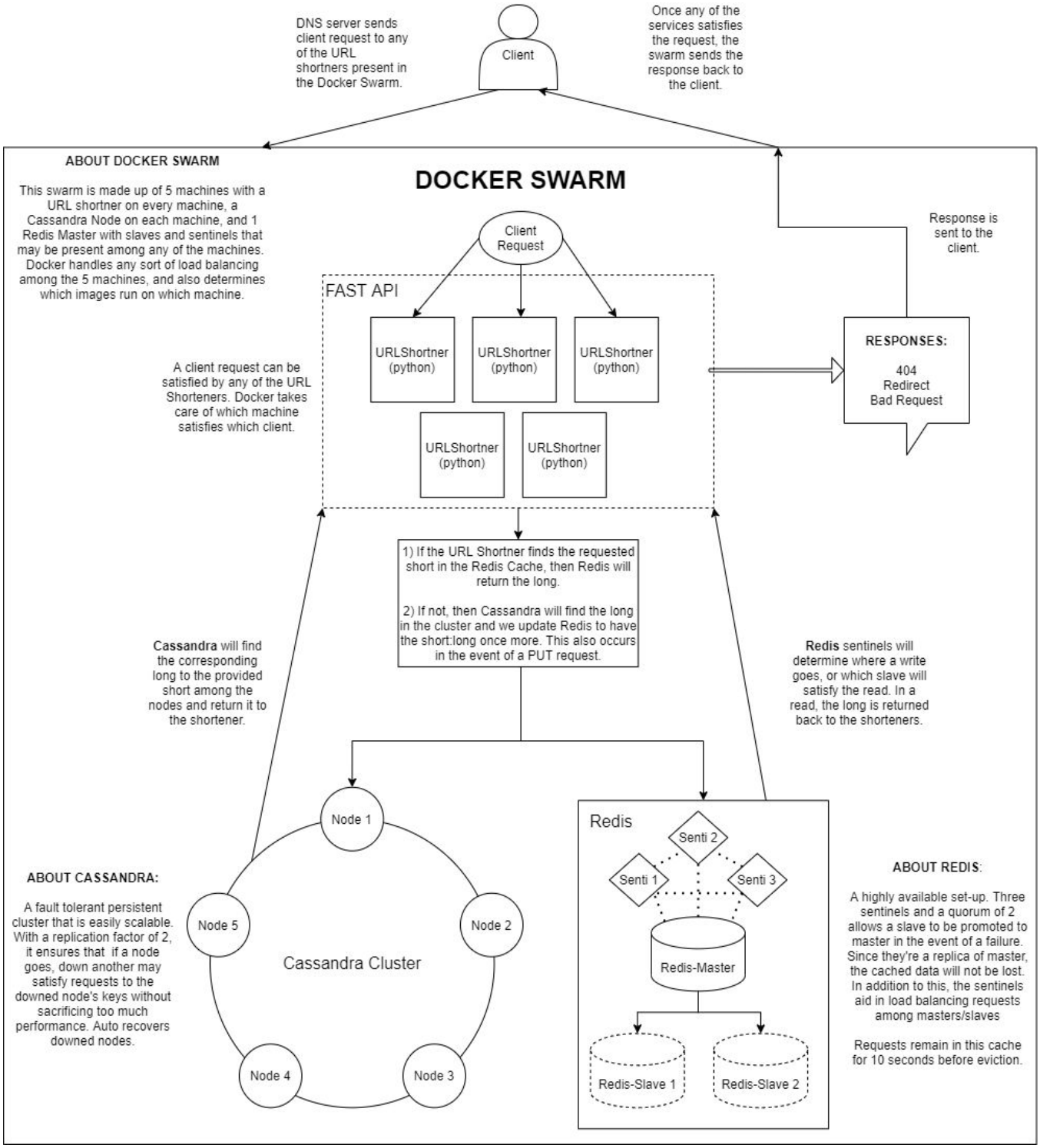
Dank URL Shortener - V2

Fast, **A**vailable, **S**calable, **T**urbo-speed - A Daniel Wang, Wan Song Lee, Liam Aiello Project



Abstract

Our goal for the DankURL Shortener V2 is to continue to focus on two primary characters of CAP theorem. Partition Tolerance and Availability, similar to V1 we want to ensure that our services can continue to operate even in partial outages or failures. Failover has been implemented in our new technology stack of Cassandra, Redis, and Docker Swarm. To implement a high performance system consistency was loosened due to Redis, however it continues to remain eventually consistent and have strong consistency. Routing and load balancing is now more fine grain and done on various components, redis-sentinel, docker swarm, and cassandra cluster. Replication is done on the cassandra cluster and acts as a cold storage for all requests that fail to hit the cache. Deployment has been made simple with docker images to ensure that all systems are able to install the correct stack and scale the services.

Hardware

- For our preliminary tests and design, the service was scaled across 5 VMs each with 1 core 1 thread and 3GB of ram.
- OS of choice was Ubuntu 16.04.7 LTS across all 6 nodes.
- Host machine is operating on an i7-9700.

Technologies

FastAPI

FastAPI is a multithreaded high performance web framework that we used for our URLShortener. Together with Gunicorn and Uvicorn, it allows us to produce production ready code. Additionally, FastAPI supports async calls for better performance during high IO operations.

Docker and Docker Swarm

Docker is a platform that uses OS-level virtualization to allow software to be deployed on any system. It does this through “containers” that bundle software, libraries, and config files to accomplish tasks, similar to a light-weight virtual machine. Docker is the “glue” of our system, and is responsible for the deployment of each service and setting up the environment they run on.

Docker Swarm is a feature of Docker that orchestrates multiple machines to run multiple containers (it creates a machine cluster). For our system, Docker Swarm is responsible for load balancing requests to the URL shorteners, as well as ensuring the health of the cluster. It is also responsible for the successful communication of these containers deployed on different machines within the cluster.

Cassandra

Distributed NoSQL Database providing high availability with no single point of failure due to replication of keys across nodes and health checks. This technology is responsible for persistence in our system, and is highly scalable by simply adding a node to the cluster.

Redis

In memory key-value data store with support for a wide range of data structures.
We used this technology as a cache for our swarm, so requests can be satisfied quickly by taking advantage of RAM memory speeds.

Gunicorn (Uvicorn)

Gunicorn is our production WSGI server for FastAPI, and communicates with our URLShortener. This is an asynchronous multithreaded WSGI server. Allowing us to handle more clients and requests during high IO operations.

Wrk (Benchmarking)

WRK is the benchmarking and performance tool we're using. It's lightweight and is built to optimally send requests to a web server from a single client. The tool spawns multiple threads simultaneously and reuses http connections to reduce client side bottleneck as much as possible

Architecture

General Overview

Everything is implemented within Docker Swarm, this enables us easily utilize docker images, containers, compose, and recoverability. This introduces a lot of automatic scalability, fault tolerance, monitoring, logging, and setup into our system. Docker allows us to create lightweight virtual containers to separate applications, while connecting them to a network overlay. Additionally, Docker uses ingress and round robin load balancing to distribute tasks to various containers starting from our URL shortener. Additionally Docker monitors various containers to see if they're down or if they could handle a request, removing single points of failures.

Once the request gets routed to a Gunicorn Server, Gunicorn then asynchronously handles the connection and routes it to the URL shorteners using FastAPI, if the request is a GET then the URL shortener checks Redis first to take advantage of its speed when it's used as a cache. If the requested short is present in Redis, then the URL shortener will simply return the desired long back to the user and redirect them to their page. If the short is not in the Redis cache, then it consults with Cassandra for the appropriate response. Using CQL, the shortener will query the Cassandra DB for the desired long using the corresponding short and if it's present, the shortener will send it back to the

client and redirect them. In the event of the short not being in either Cassandra or Redis, a 404 not found is returned to the client. If a PUT request is made, both Redis and Cassandra are written to, so after a PUT request both the cache and the DB will have the short to long mapping. The client is then notified that their mapping has been successfully documented. Should the write fail, an error is returned to the client.

Redis Architecture

Our system features a Master-Slave-Sentinel Redis Architecture that has one master node, two slave nodes, and three sentinels that monitor the master. The sentinels are configured to have a quorum of two, so two must agree in order to promote a slave to master in the event of failure. We have chosen to do this to provide high availability, as without the sentinels present if the master fails, a slave will not be able to be promoted to the next master, causing the cache to fail. In addition to this, the sentinels also assist with write and read load balancing across the redis system, as they will choose which slave nodes satisfy reads as well as which master nodes propagate certain writes (in our case, it will always be the single master node), lowering the chances for hotspots within Redis. Finally, having a master-slave system gives the system a large boost to our read performance since the slaves will take care of reads. Since most requests will be reads, it is important to optimize these over the writes. A downside to our Redis architecture is the fact that the master will now have to propagate two writes (one for each slave) instead of one, effectively doubling the number of writes we do in Redis. We have chosen to take this trade-off as it is vital for this cache to be highly available, since it is used across the system.

Cassandra Architecture

Cassandra handles our cold data store and persistence. A cassandra cluster with a replication factor of 2 was implemented to ensure fault tolerance and availability. All data is replicated via default hash mapping, and in the scenario a node goes down, the cluster will forward the request to the next replica. This ensures that we have high availability and probability of not serving data is the probability of both the master and replica going down. The additional benefit of this is that data is sharded, so that we can increase the maximum volume of our key-value store simply by scaling the number of nodes in the system. Additionally, Cassandra is a key-value noSQL store making it the ideal database for an urlShortener. A benefit of using Cassandra Docker is that when a node does go down, it’s able to revive and be brought back into the Swarm and Cassandra cluster automatically. This is because cassandra clusters identify members based on HOST-ID which is stored in it’s persistent volume.

Running The System

Starting the Swarm

docker swarm init --advertise-addr IP1
./deploy

Will automatically build any images we have and deploy the services to the nodes in the swarm.

Stopping the Swarm

./undeploy

Removes the service stack from the swarm.

Scaling the Swarm

ssh IP 'docker swarm join --token .... IP:2377'

Additionally, services configured to global will automatically start new replica services on the joined node.

Removing a Node from the cluster

ssh IP 'docker swarm leave --force'

Testing The System

Service Monitor

The system has a service monitor located on localhost:8080 on the manager node. Here you can find the statuses of every container running on the swarm and get real time status updates. This utilizes the docker swarm visualizer image, which spawns a separate service to monitor all memory usage and uptime.

Performance Read Tests

Using WRK, we can easily minimize client bottlenecks. The following exert spawns 12 threads, 300 connections to work. The threads are context switching during high IO operations which makes this optimal on high IO services. Duration via -d is set to 1 second. Additionally we can configure this to show latency.

```
wrk -t12 -c300 -d1s -s put.lua
“http://127.0.0.1:4000/?short=gg&long=google.com"
Running 1s test @ http://127.0.0.1:4000/?short=asdf&long=asdf
  12 threads and 300 connections
```

```
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   145.93ms  102.85ms 656.66ms   81.93%
    Req/Sec   156.21     46.79   242.00     76.67%
  1872 requests in 1.06s, 303.47KB read
Requests/sec:   1762.12
Transfer/sec:    285.66KB
```

Performance Write Tests

Using WRK, we once again minimzie client bottlenecks. The following exert spawns 12 threads, 300 connections to work. The threads are context switching during high IO operations which makes this optimal on high IO services. Below is a 15s test run. Additionally we can configure this to show latency.

```
wrk -t12 -c300 -d15s http://127.0.0.1:4000/gg
Running 1s test @ http://127.0.0.1:4000/gg
  12 threads and 300 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    45.54ms   51.20ms 406.31ms   90.90%
    Req/Sec   662.69    104.83     1.05k    75.93%
  7605 requests in 1.05s, 1.42MB read
  Non-2xx or 3xx responses: 7605
Requests/sec:   7268.39
Transfer/sec:      1.36MB
```

Fault Tests

We kill various containers (such as the redis-master or a Cassandra node) and observe how the swarm recovers through commands such as `docker service ls` and `docker service logs` to see how the container reconnects to the service/cluster. We would also try doing both GET and PUT requests when these containers were down to verify fault tolerance.

Logging

To monitor logs generated by our server, we can run `docker service logs a2_stack_web | grep LOGGER:`

More fine grain monitoring can be done on a container to container basis via:

`docker container logs a2_stack_web | grep LOGGER`

Cassandra Monitor

Nodetool status

- Monitors the status of our cassandra cluster, partitions, and etc

Nodetool Info

- Displays node info, uptime, and etc.

Analysis

Vertical Scaling

Docker automatically allocates resources to containers, thus as hardware improves containers will be able to automatically be allocated more resources. Memory and CPU are automatically allocated. Additionally, services such as URLShortener gain improved performance from more cores or processing power, as FastAPI and Gunicorn Server are multithreaded technologies. Redis would benefit from having more memory before having to LRU evict data. Cassandra would benefit from increased volume sizes

Horizontal Scaling

We can scale up/down the number of web servers, cassandra, and redis nodes by adding/removing hosts to/from the swarm. Additionally fine grain service scaling can also be done by `docker service scale` that do not implement global mode. Global Mode was opt'd for deployment on some services that we wish to scale 1 container to 1 Server/Node. As scaling increases, new services would spawn allowing us to handle an increased amount of requests automatically.

Caching

Caching is handled by Redis Cache Servers, storing requests in memory to quickly provide responses from cache instead of having to hit a cold disk. Redis slaves are replicated to provide increased read performance, with sentinels to load balance. Additionally, data is expired to ensure that stale data does not remain in case of disaster.

Availability

We have configured the `docker-compose.yml` to recover containers in the event of failure, and since there are multiple instances of Cassandra, Redis, and URL Shortener, there is no single point of failure in the system. In particular, our redis setup shows this by utilizing a master-slave redis system with sentinels. Redis has high availability this way, since in the event of a crash on the master node, one of its slaves will be elected to be the new master. Cassandra has its own way of exhibiting high availability, as key replication ensures that other nodes in the cluster will be able to satisfy requests if one of the nodes fail. Finally, Docker Swarm will automatically attempt to recover crashed containers and even nodes before bring them back into the swarm.

Consistency

Our system is *eventually* consistent, since the only way for Redis to be consistent is to have one instance. This is not the case in our system, so there are no guarantees for

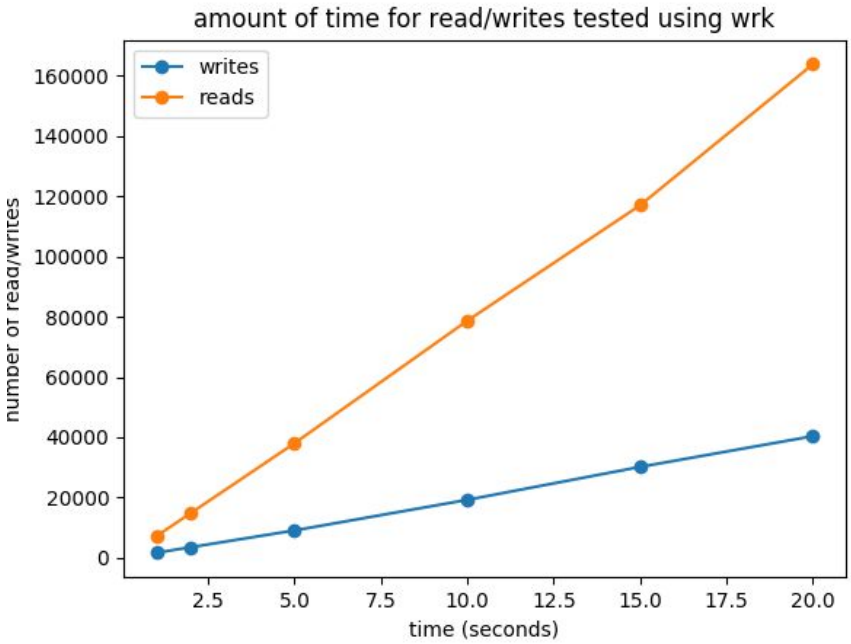
consistency in our URL shortener. However, redis with master-slave offers strong consistency, eventually becoming consistent. See https://redis.io/topics/replication and https://redis.io/topics/sentinel.

Monitoring UI:

We used the visualiser image to see the status of our stack and check if any of the containers are down.

We can also monitor the state of cassandra by running `nodetool status` on one of the containers running cassandra

Performance



We used the wrk tool to test our system and managed around 6000-8000 GET requests/second and 1500-2500 PUT requests/second.

Put Requests

```
wrk -t12 -c300 -d1s -s put.lua
“http://127.0.0.1:4000/?short=gg&long=google.com"
Running 1s test @ http://127.0.0.1:4000/?short=asdf&long=asdf
  12 threads and 300 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   145.93ms  102.85ms 656.66ms   81.93%
    Req/Sec   156.21     46.79   242.00     76.67%
```

```
  1872 requests in 1.06s, 303.47KB read
Requests/sec:   1762.12
Transfer/sec:    285.66KB
```

Get Requests

```
wrk -t12 -c300 -d15s http://127.0.0.1:4000/gg
Running 1s test @ http://127.0.0.1:4000/gg
  12 threads and 300 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    45.54ms   51.20ms 406.31ms   90.90%
    Req/Sec   662.69    104.83     1.05k    75.93%
  7605 requests in 1.05s, 1.42MB read
  Non-2xx or 3xx responses: 7605
Requests/sec:   7268.39
Transfer/sec:      1.36MB
```

Through WRK, we can see that our average read latency is approximately 45ms and average put latency is 145 ms. This is due to reads hitting redis cache, while writes having to hit write to both Cassandra and Redis. We opted to test requests that can be handled per second rather than testing how long it takes to handle N requests in order to show how our system performs over time and check for bottlenecks. As per our expectations our system scales linearly with time.

One of the great things about our implementation is read speeds, we assume that URL shorteners will typically have more reads than write, so replicating a read only cache to all URLShorteners to access increase the performance significantly.

A downside however, is that the redis master node does need to propagate the write data to all replicas/slaves. As slaves scale, we would also have to scale master, which would in turn fragment the cache. As of current, we opted for 1 master 2 slaves to handle all requests to cache. There is no good option to automatically scale as decisions to fragment the database and scale slaves will have to be done on a case to case basis.

Additionally, GET requests that do not hit cache will be substantially slower. Approximately the same as the write speeds. However, upon getting from disk our implementation will then add its to the redis cache, evicting via LRU if full.