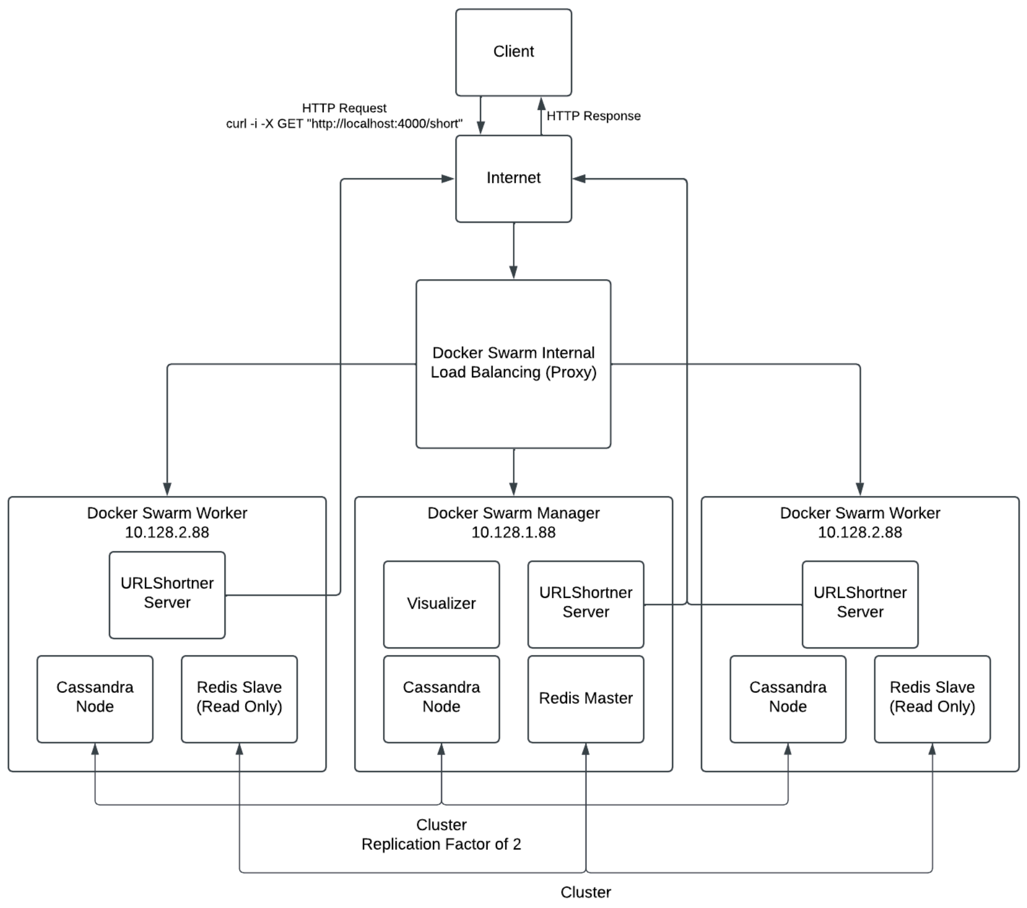
CSC409 Assignment 2

ganhzale, farre176, chibingb

TABLE OF CONTENTS

Architecture



Our system architecture is built on a docker swarm setup. The swarm consists of two worker nodes and a manager node, which will receive the client requests from an internal load balancer. Within the swarm, we deploy three independent web servers which handle the initial GET/PUT requests. Depending on these requests, they are either routed to the redis node, which is used for caching, or the Cassandra node which stores and retrieves client requests.

Redis implements a Master-Slave configuration where any new write requests are handled by the redis master and the slave nodes are read only. This setup ensures eventual consistency and allows for efficient scaling of read operations across the system.

Cassandra consists of three nodes inside the swarm, utilizing consistent hashing to partition data across the cluster. To enhance data fault tolerance, each partition is replicated once, resulting in a replication factor of 2. This replication ensures that in case of a node failure, the data is still available on another node, thus the system availability is unaffected.

Proxy

In our docker swarm based architecture, we rely solely on docker's built-in load balancing rather than using a dedicated proxy server like in the previous assignment. Docker swarm automatically manages the distribution of network traffic to our python web server replicas, reducing the need for an external proxy.

Caching

In this setup, we are caching with Redis, where each replica of the web server has its own local redis node. Redis implements a Master-Slave architecture, where the slaves are read only and the master can write and read. This architecture allows all nodes to read from the cache efficiently, while any updates (e.g., handling PUT requests to map a new long URL to the short) are processed by the master node and propagated to the slaves, ensuring data consistency across the cache.

This will improve performance compared to our last setup, as there are now three instances of the cache, as opposed to a single instance on the proxy server. This setup should mean the response times are significantly reduced now for all cached GET requests, thus improving availability of data.

Load Balancing

In our system, load balancing is managed directly by the docker swarm, with one swarm manager orchestrating tasks across the two workers. By giving each service inside the swarm a virtual IP address, each request can be routed evenly across the replica network. This configuration allows all the active webs server instances, spread across the four virtual machines to receive a distributed load of incoming traffic.

Although our implementation for load balancing in A1 was effective at distributing evenly, docker swarm should be much more optimized, especially when scaling up or down. In our previous setup, scaling involved halving the hash ranges of an existing server to allow a new server to take on half of that load. Although this would help the load of this one specific server, the load between the remaining servers does not get redistributed evenly. However, docker swarm can dynamically distribute the load, ensuring that adding a new server will redistribute traffic evenly across all active nodes. This ability to seamlessly scale up, will improve performance and reduce bottlenecks effectively when under a higher load compared to the previous model.

Fault Tolerance

Our system achieves fault tolerance by deploying the web server in docker swarm with a replication factor of 3, ensuring that multiple replicas of the web server are distributed across the swarm. By specifying the mode to global in the docker compose file, it meant that each node in the swarm will run one instance of the web server. This meant that if one web server goes down, we still have two more servers, as the software has been replicated. In addition, the docker swarm will automatically attempt to redeploy the lost instance so that the level of fault tolerance will remain the same.

To ensure data is also fault tolerant, we run Cassandra inside the swarm and each cassandra data partition is replicated on another node. Cassandra's replication mechanism ensures that if a node fails or is terminated, the data remains accessible via another replica stored on a different node. This replication strategy prevents data loss, as the requests can be seamlessly directed to the replication node instead to maintain availability.

Healing

If a service failure occurs, docker will attempt to bring the service back online. For example, if an active Redis service fails, docker will attempt to reconnect the service to the swarm. However if the whole node fails, meaning the web server, redis instance and cassandra failed, we would have to bring up a new node manually and monitor this manually too.

Monitoring System

Docker is stateless so it doesnt need to know its own state. Docker monitors its active services, if one goes down, the monitoring system detects and attempts to bring up a new service in its place, which has the same state as the previous service.

Monitoring UI

Although we do not have a dedicated monitoring UI, docker has in-built features that makes it is manageable from the command line. To test if services are running, we used docker service ls, and docker server ps to monitor these states.

Also because of docker's self healing, there is not much need for a monitoring UI, as any issues are handled automatically by docker itself.

Availability

The system is designed with high availability in consideration, thanks to how we configured our services. In Cassandra we set the replication factor to two, ensuring that each data partition is duplicated on another node. This configuration guarantees that if data is lost on one node, it remains available on the replicated node, preserving the data accessibility.

The Redis and Python web servers are replicated 3 times, where each python web server will have its own instance of Redis. Redis is configured so that one node is a master node and the rest are slave nodes. All data is synchronized from the master to all slave nodes ensuring consistency and availability across the cache. However, if the master node was to fail, the cache could not receive new write requests. On the assumption the master node remains operational, the system availability would be unaffected because all the nodes can still return a valid cache response.

The python web servers will also remain available, if one server fails. Remaining instances continue to handle client requests without being interrupted. Therefore even if an isolated server failure occurs, there is still a guarantee the software remains fully operational.

Consistency

Our server uses Redis for caching and Cassandra for database storage. The consistency of our server is heavily influenced by the characteristics of these two technologies. Both Redis and Cassandra follow an eventual consistency model, which means they prioritize high availability and partition tolerance over immediate consistency.

While Cassandra provides configurable consistency levels, we have opted for the lowest consistency level to maximize availability. For Redis, we implemented a master-slave configuration to reduce the load on the Redis master by offloading read requests to the slaves. This setup may lead to consistency issues during failovers and replication delays.

Failure Testing

In our orchestration folder, we have a kill bash file, which simulates the failure of a server. After executing this file, the system still performs seamlessly. This is because all the data in the cassandra db is replicated by a factor of 2 so no data is lost. We would also lose a web sever and redis node, but assuming the master node does not fail, the system software remains fully operational.

Load Testing

We have implemented the same load testing as for A1 in A2.
We have used three types of tests, GET requests, GET requests with caching, and PUT requests. Throughput and latency under various loads are demonstrated using graphs in our performance section.

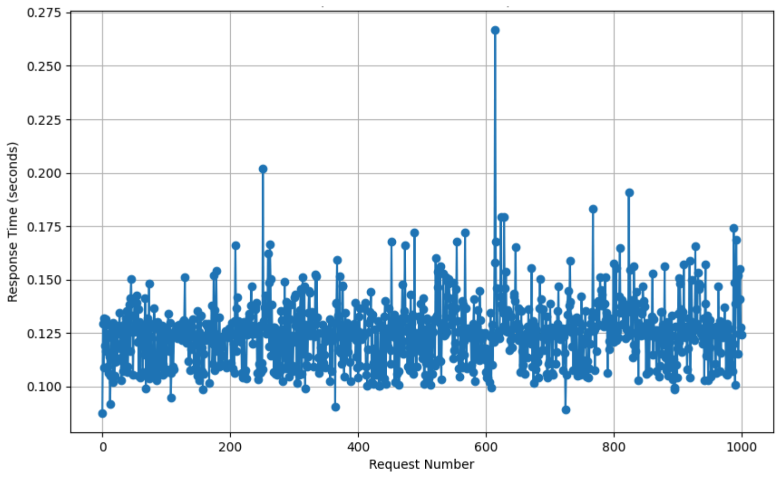
Bug Testing

To test the main features of this system, we made GET and PUT requests using cURL. This allows us to track where each PUT request would go to, by then checking the database table in Cassandra.
This feature allows us to see if data is correctly being replicated, and partitioned across the cassandra nodes. When using the command line to execute cqlsh instructions, we could identify replicated data across two of the three nodes. This would demonstrate a replication factor of 2. The conducted tests also showed us that data was being partitioned correctly, as the cassandra nodes did not all have the same data.

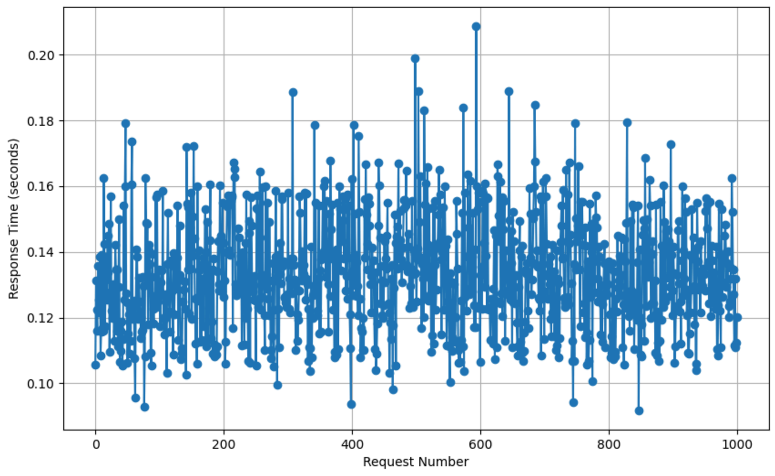
To test redis, we set up print statements to show if there was a cache hit or miss. By using curl, we could see when the cache was being used after a repeated GET request was made, and when the cache was updated if the long value for a short url needed to be overwritten.

Performance

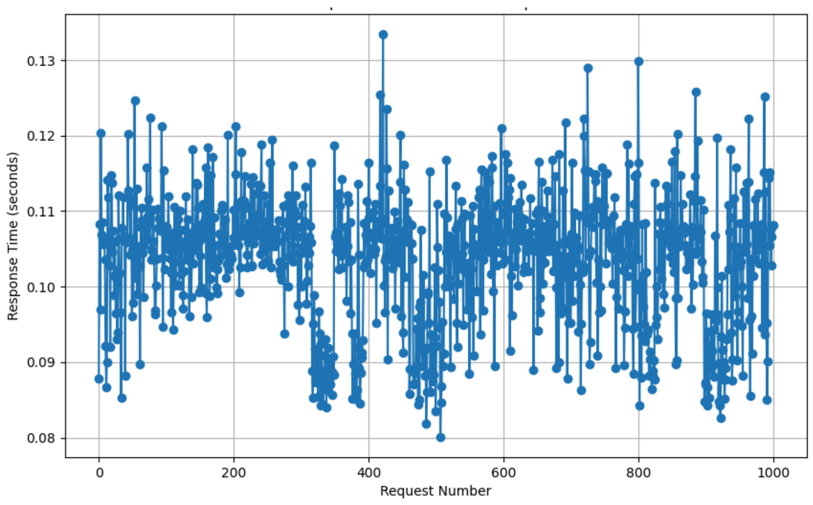
GET Request Cache Response Time



GET Request Response Time



PUT Request Response Time



GET and PUT requests analysis

Our analysis shows that the system handled 4000 GET requests with an average of 7 seconds with caching enabled and 11 seconds without. This improvement demonstrates the effectiveness of the Redis cache being utilized, as the time to retrieve most recent data resulted in significantly faster response times.
Similarly to our GET requests, 4000 PUT requests took 10 seconds, which shows that storing and retrieving data from Cassandra takes roughly the same time. This emphasizes the importance of caching, as reading and writing to Cassandra is clearly a costly operation.

In comparison to our previous results, the requests per second is much lower than before. This is probably due to the shift from Java to Python. Therefore, it is difficult to see the true benefits of using external services like Redis and Cassandra, when we are using a slower language.

In a future comparison, we would have executed the web services using Java to see if there is a noticeable difference when using Redis and Cassandra over our own implementation.

Data Scalability

The current architecture of the system is designed to support data scalability (adding more space). If the primary storage is increased, the Redis cache can store more frequently accessed data, reducing the need to query the Cassandra database. If the secondary storage is increased, cassandra can store more records (short URLs and long URLs). This ability to accommodate for larger data volumes as the demand increases means our system can scale smoothly without having to reconfigure the architecture.

CPU/Node Scalability

Our system supports both vertical and horizontal scalability, allowing for flexible performance when scaling up. Vertical scalability, which involves increasing CPU power will increase processing capabilities. This means each node will be able to handle requests quicker, improving the overall speed of data loading, storing and the response times for each client request. Therefore, by scaling the CPU power, we can reduce latency for each user which will be particularly noticeable when the system is under normal or lighter nodes. However, as traffic increases, there comes a point where even enhanced CPU power cannot prevent bottlenecking, since individual CPU capacity has its limits.

If the system is under peak traffic, it can help to scale horizontally, which involves adding more nodes to the current cluster. By increasing the number of nodes, the system can better distribute the traffic, meaning the load on each individual node can be minimized. This redistribution of traffic can significantly reduce the effects of bottlenecking and ensure no nodes get overwhelmed when under the highest amount of traffic.

Admin Scalability

Docker swarm allows us to easily add new nodes to the system, enabling each service to dynamically scale up without being detrimental to system performance. For example, docker swarm handles redistributing the load across the nodes ensuring all resources are being utilized efficiently. Cassandra handles adding a new node to its current configuration by using consistent hashing, which seamlessly partitions and redistributes the data to maintain the balance across all nodes. Redis handles adding a new node by creating and synchronizing a new slave node. Finally, the python web service is configured so any new instance operates identically to the existing web services, handling requests to its redis instance or routing requests to the cassandra database when needed. Each of these services are optimized to reduce downtime and maintain system availability.

Orchestration

To start the system, we use a doAll orchestration file. This initiates the docker swarm and launches all the docker services in the system including, Redis, Cassandra and the Python web servers.
We have another bash script to add a new node to the system, and this will create a new python web server which operates identically to the existing web servers. It also deploys a redis instance which will be configured as a slave node receiving the same cached data as the other slaves to maintain consistency. Finally a cassandra instance is created that will receive a replicated data partition from another node and its own data partition to ensure availability across the cluster.

We also have a bash script to delete a node. This is used to simulate the effects of a node failure.

Finally to stop the system, we have a killAll bash script which will terminate all the services in the docker container, and cause all nodes to leave the swarm.

Data replication ensures that in the event of a catastrophic failure or complete server outage, data can be recovered from the replica server without losing critical information.

Monitoring system

Monitoring is executed by swarm. Docker handles most of the Monitoring in our system. If the web server or cassandra goes down, docker will detect it.