CSC409 Assignment 1

ganhzale, farre176, chibingb

TABLE OF CONTENTS

System Architecture Representation

1. General system application

The application system consists of 3 Sqlite servers and another server which acts as the proxy and Sqlite server. The general diagram (see Figure 1.) provides an abstract overview on the whole system, demonstrating the interactions between the client, proxy and sqlite servers. The core components that underpin this system will be discussed in further sections.

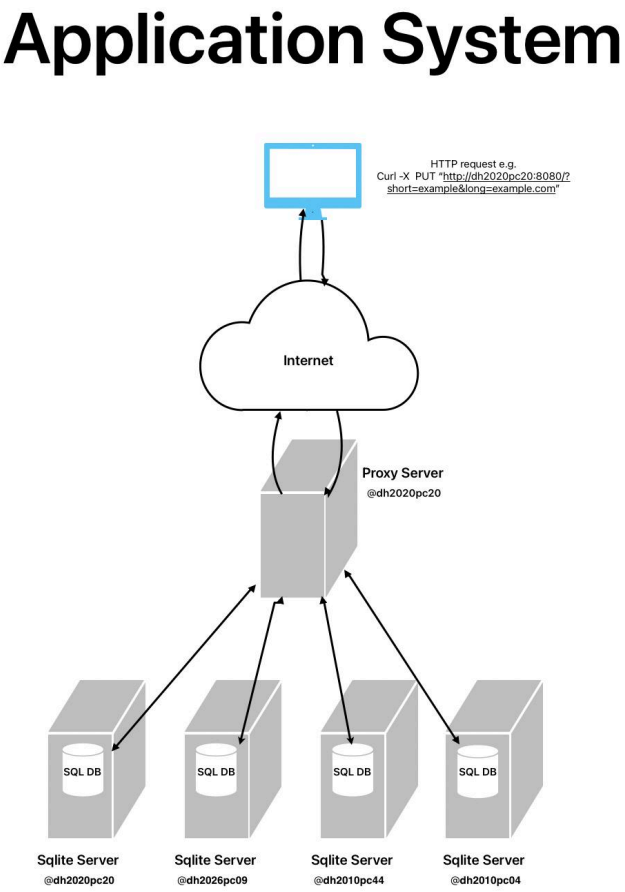


Figure 1. General Application Diagram

2. Data flow

The following diagram introduces the general flow of data in our system. It involves clients creating GET/PUT requests to the proxy server, which are then distributed across the sqlite servers. This report discusses the details of how data gets distributed, and how the flow of data is affected when the system scales up (adding servers) or scales down (terminating servers).

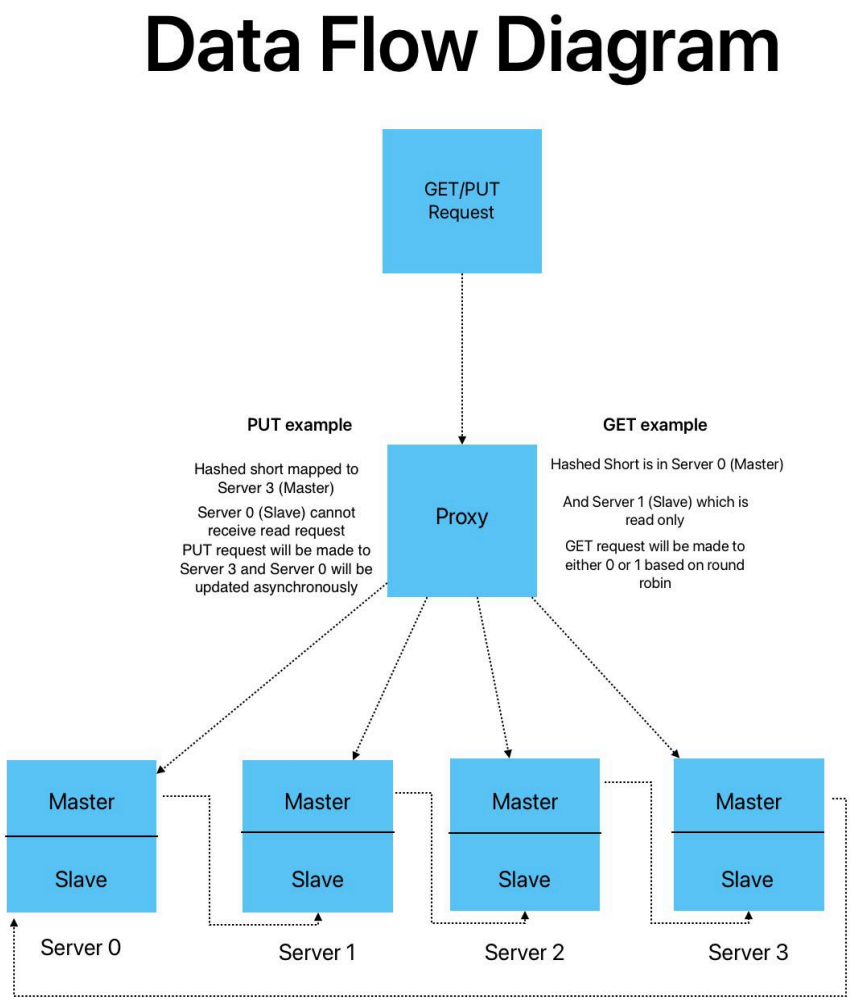


Figure 2. General Data Flow Diagram

3. Monitoring

As illustrated by the diagram, the monitoring tool makes a GET request to the server and if it replies then it knows that it is alive, otherwise it logs the server failure for the self healing script to handle.

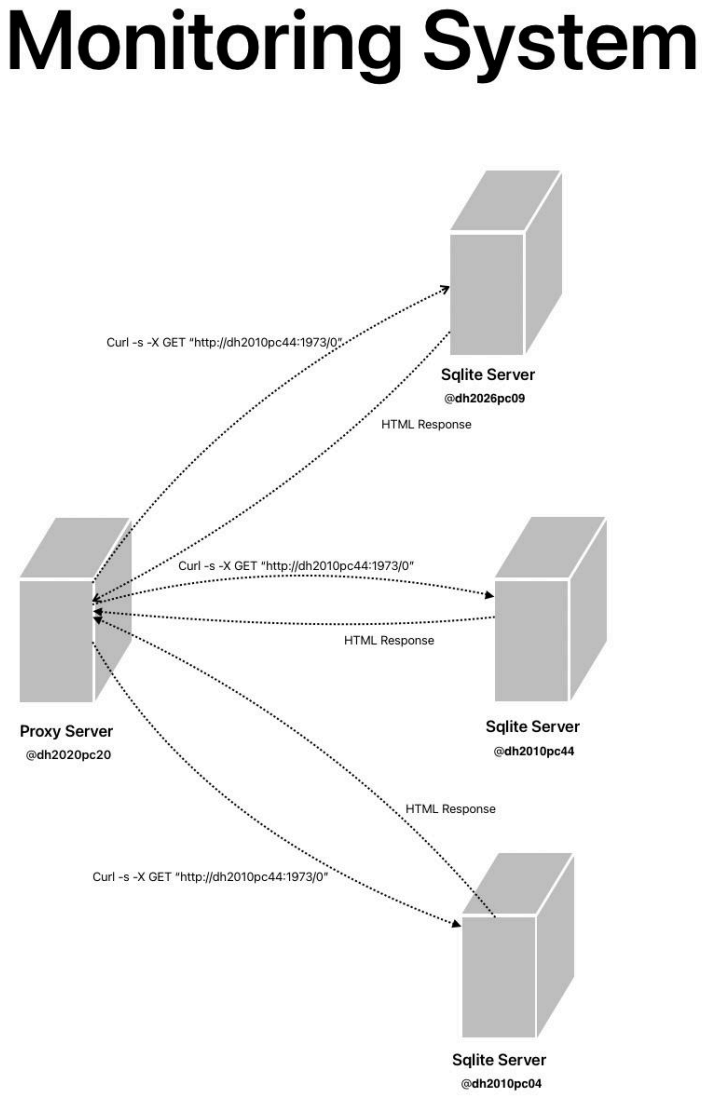


Figure 3. Monitoring system diagram

Balancing CAP Theorem Tradeoffs

1. Consistency

Our system provides a high level of consistency by using safe multithreaded servers and caches. Moreover, we have designed an architecture similar to redis to ensure the data replication remains consistent.

To implement this we used a “Master-Slave” relationship between the primary server and the replicated server. In other words, the master server can receive read and write requests but the slave server can receive only “reads”. If only one server is creating the write requests and the slave is synced to the master, there would be no inconsistency in the data between them.

There is one exception where this solution is not strongly consistent. If the client attempts to write to the master server and reads immediately after, the data may not be found if the GET request is sent to the slave node. There is a small time-window where the master data is being asynchronously copied to the slave database, so the request will not be found, creating an inconsistency. We decided this would be a rare case and a small tradeoff, at the cost of replicating the data and maintaining availability.

2. Availability

The following list of aspects covers our architectural choices that we made in order to assure a high level of availability in our system:

- **Proxy-Based Design**

  The proxy acts as the central access point, directing client requests to the appropriate Sqlite server or replica. It can handle failures by rerouting requests to another server if the primary one is down to maintain availability.

  The proxy can also periodically check the health of the servers and reroute requests to available replicas, ensuring availability even in the case of partial system failures. On the other hand, the proxy is a 1 point of failure in the whole system so if it fails the availability of the system becomes 0.

- **Multithreaded Proxy**

  The multithreaded nature of the proxy allows it to handle multiple client requests concurrently, increasing the system's throughput and reducing the likelihood of bottlenecks, therefore improving availability.

- **Replication Factor of 2**

  Each data partition is stored on two different servers, a primary and replica. If one server goes down, the data can still be accessed from the replica, ensuring availability even during a failure.

- **Equally Partitioned Data**

  Data is evenly distributed across servers, ensuring that no single server is overloaded. This balance helps maintain high availability under normal operation by distributing requests evenly.

- **Caching**

  We decided to implement the caching on the Proxy. This decreases response time for each of the GET requests. It's a tradeoff for the speed but we are limited by the size of the RAM in the proxy. In addition, cache works with FIFO principle that ensures that availability of the most recent requests is assured at faster speed. We discuss caching more in the "caching section"

3. Partition tolerance

The data is distributed into 4 equal partitions across the 4 servers, which minimizes the impact of failure by confining it to a smaller portion of the overall data. By replicating each partition across multiple servers, the system ensures that if one partition becomes isolated because of a network failure, the replicated data on a backup server is still accessible.

The proxy continuously monitors server availability, and in the event that a server or partition becomes unreachable, the proxy automatically reroutes requests to the available replica on another server. This load balancing mechanism ensures that client requests are rerouted without interruption, despite the network partition.

Key Principles of Distributed Systems

1. Data partitioning

As shown by the diagram below, each server has an equal partition of data so on average if 1000 PUT requests are made, each server will receive an even distribution of the requests (as the default java hash() function has a very even distribution).

Hash collision doesn't affect our system because hashes are just for partitioning and load balancing, so If 2 requests have the same hash, all it means is that they are both on the same server.

The hashing of the "shorts"is implemented by using the inbuilt java hashing function which gives a hash value in the range of 0-9999. From the diagram you can see that each server gets a quarter of the hash range, so the data is distributed evenly across the servers. This results in an even load when requests are being made so not only is the data being partitioned but it also acts as a load balancer.

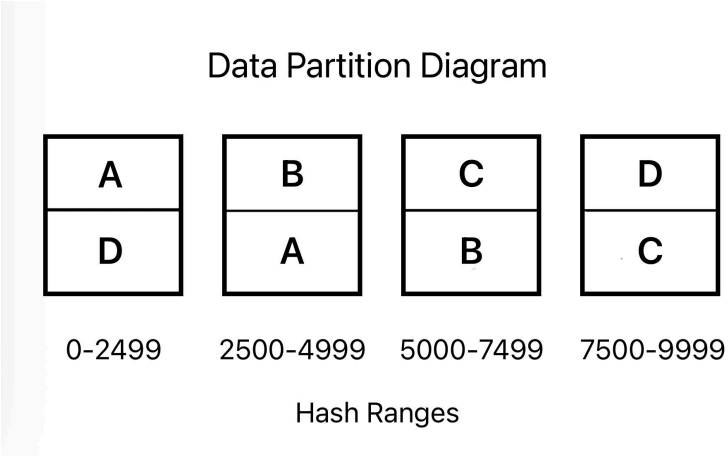


Figure 4. Initial hash ranges with 4 serves

2. Data replication

When replicating data, we decided on a replication factor of 2. This ensures fault tolerance by allowing the system to recover from a single server failure while keeping the system fully functional. Since each server holds a quarter of the total data, replicating the data on two servers guarantees that no single point of failure can lead to data loss. A higher replication factor would provide diminishing returns, as it would unnecessarily increase storage and network overhead without improving fault tolerance for a system of such a small size.

Replication Implementation

As discussed in the section on consistency, we implemented a Redis-inspired replication architecture. This setup includes a master server, which can perform both read and write operations, and a slave server that is read-only. Data updates are handled asynchronously by threading an I/O stream from the master server to the slave. This design allows the original write request to complete without waiting for the slave to finish copying the data, thereby improving overall system responsiveness and allowing the master server to efficiently manage concurrent requests.

Architecture Benefits

- **Improved Read Availability**
  Read operations can be load-balanced between the master and slave servers which should improve the overall read performance whilst reducing load under high traffic conditions.

- **Disaster Recovery**
  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.

- **Minimized Downtime**

  Replication helps minimize downtime during planned maintenance or server upgrades, as replicas can take over while primary servers are being updated.

3. Load balancing

Hash-Based Implementation

- Each incoming request is processed by applying a hash function to the short. This hash value is then used to determine which server will handle the request.
- The hash function ensures that requests are consistently mapped to specific servers, which allows for efficient distribution and minimizes the likelihood of overloading any single server.

Architecture Benefits:

- **Even Distribution of Load**

  Hashing allows for a more uniform distribution of requests among available servers. This helps prevent situations where one server becomes a bottleneck while others are underutilized.
- **Scalability**

  As new servers are added to the system, the hash function can adapt to include these new nodes, allowing for seamless scaling without significantly impacting performance. More details in the further section where we explain the new node introduction.

4. Caching

Implementation

Our caching implementation utilizes Java's ConcurrentHashMap, which provides a thread-safe mechanism for storing and retrieving cached data. This ensures that multiple threads can read from and write to the cache concurrently without causing inconsistent data.

The ConcurrentHashMap does this by dividing the map into segments, allowing concurrent access to different segments. This results in minimal conflicts and improves performance in a multithreaded environment.

Eviction Strategy

For cache eviction, we chose the FIFO (First In First Out) eviction strategy for our cache because of its simplicity and efficiency, especially when using ConcurrentHashMap and multithreading. FIFO means the oldest cached entries will be evicted first when the cache reaches its capacity. This ensures that stale data, which has likely been less recently accessed, is removed to make room for newer data.

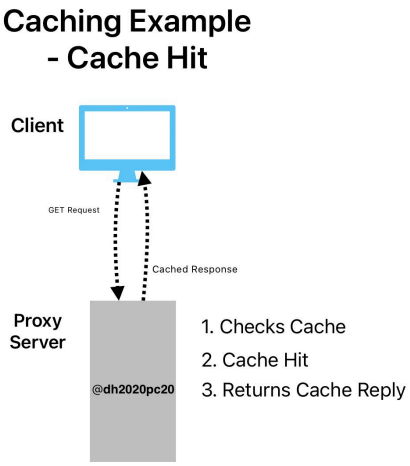
As opposed to our previous eviction method (random eviction), FIFO is more efficient.
When randomly getting and removing a key from an array list in java, worst-case time complexity will take O(n) to shift all the items to shift and take the removed keys place. By using FIFO in a LinkedBlockingQueue, we can dequeue the first item to be added, in O(1) making the eviction method faster, and have a higher chance of a cache hit.

Cache Benefits

- **Performance and Load reduction**

When a client makes a request that has been previously handled, the proxy can check the cache before attempting to connect to the backend servers. In the case of a cache hit, as

shown in the diagram, the proxy can return the cached response much faster than if it had to hash the short URL and retrieve the data from one of the servers.



Additionally, caching enhances performance for new requests by alleviating some of the computational load on the SQLite servers. Thanks to the multithreading capabilities of the proxy, it can reroute more new requests to the backend servers while simultaneously handling cached responses. This effectively improves the overall system performance whilst reducing load on GET requests.

5. Orchestration

Our scripts in the "/orchestration" directory support the following list of functionalities that help us to manage, debug and create the servers.

We can:

- Start all the servers with the ./doAll
- Kill all servers with ./killAll
- Monitor with the ./monitor
- Kill a particular server with the ./kill.bash $server
- Heal servers with ./healingServers
- Copy data with a particular time frame from 1 server to another with: ./copyLogs $server1 $server2 $timestamp1 $timestamp2

- Introduce a new server with the ./addServer

6. Healthcheck

General mechanism

Our system supports self-healing. It works by keeping track of the servers that have failed and according to timestamps.
The monitoring script inserts into a file the servers that went down. And keeps them in one instance with the first timestamp. Meanwhile another healing script reads the file and heals the servers. When the server is healed it erases position from the file.

Healing Scenario for Server Failure

1. Server fails.

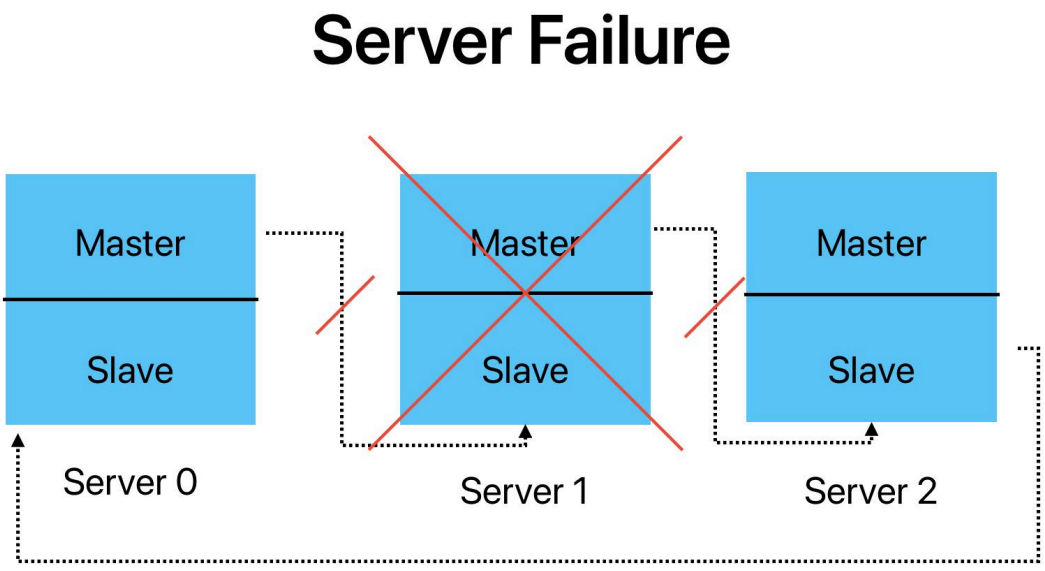


Figure 5. One of the servers went down

2. Forwarding new incoming requests
    a. The requests and according data that the node was a Master of is now sent to the Slave.
    b. The “slave data” and the accorind get request are no longer sent to the server that is down. All the GET requests are sent to a Master only.

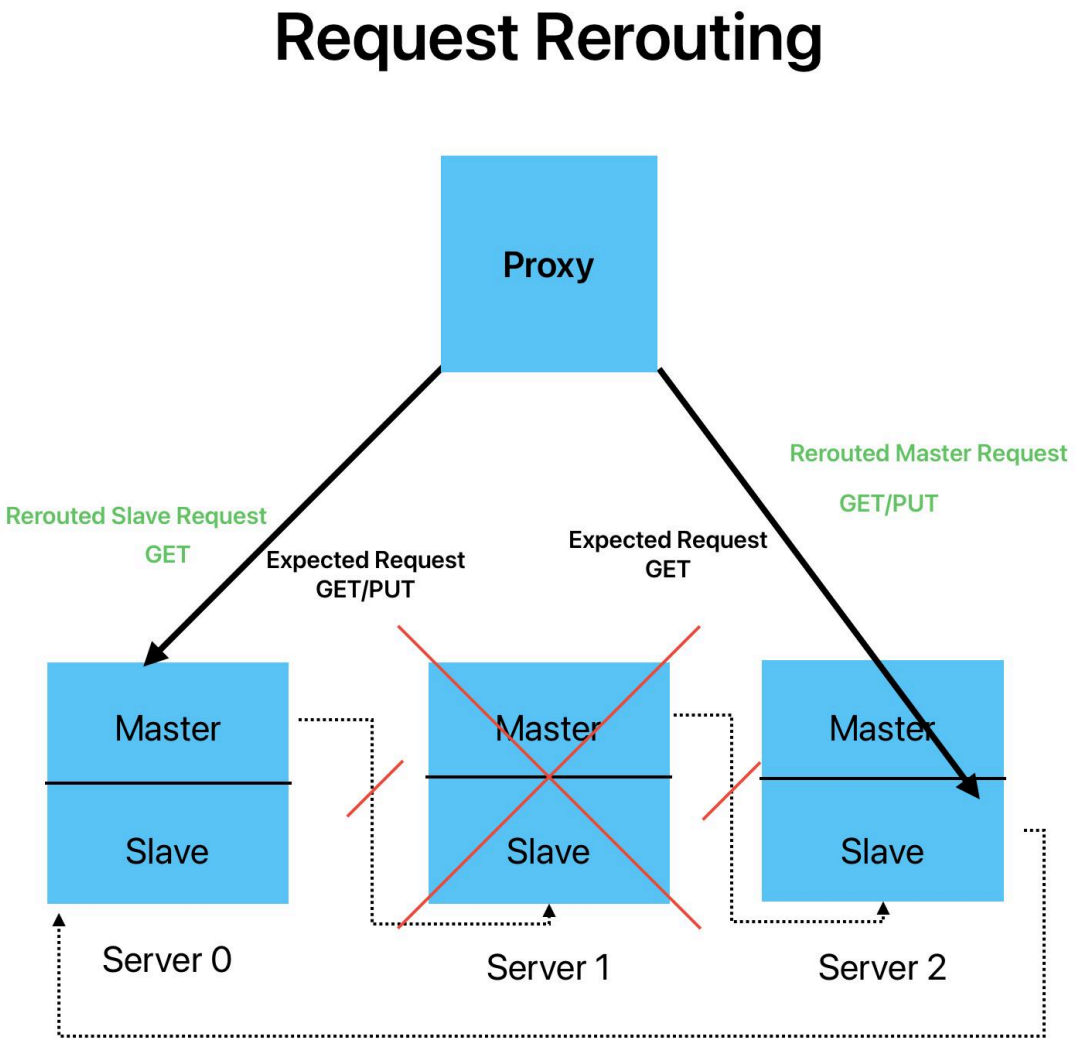


Figure 6. Client requests are being rerouted

3. Server is back up

When the server is back up we copy the according data from the next and the previous server. The master becomes a master of its own data and a slave of the previous neighbor.

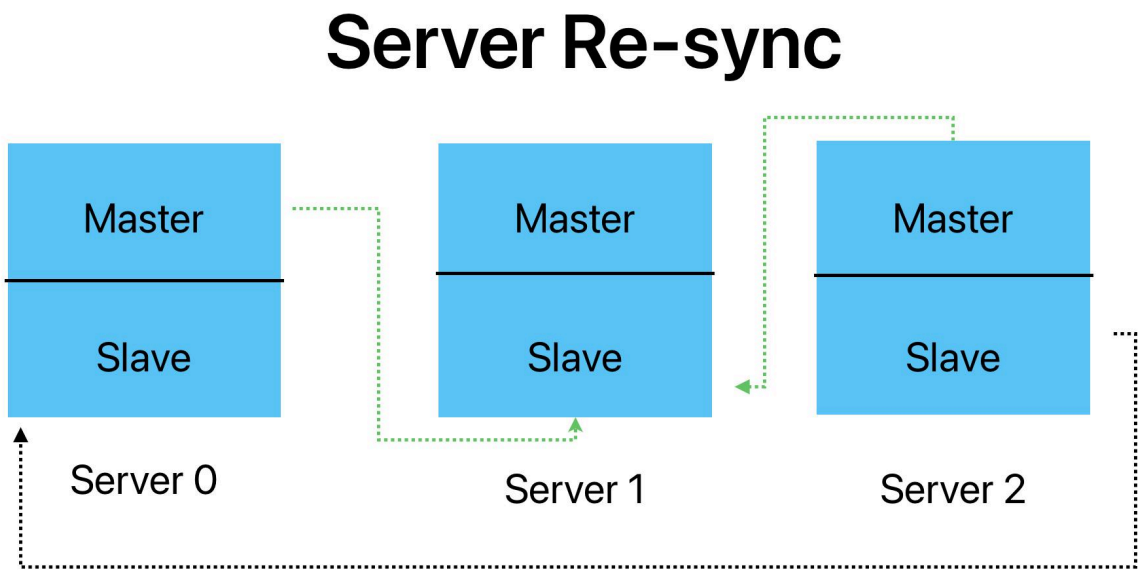


Figure 7. Copying the “downtime” data into the recovered server

A nice feature that it supports is that it copies only the data that went into the neighbors while the server was down. This removes necessity in copying extra data that could be incoming while the lost data is recovering.

To implement this we had to introduce timestamps. Sqlite has a built-in feature that writes a current time when the position was updated last time. The issue we had was that it was in UTC and our lab machines were in ETC. We had to synchronize the time to properly recover the requests.

7. Disaster Recovery

Even if all the servers are down the healing script will get them up as soon as possible. This proves a great recovery and that it's not when just one database server is down the system could be healed but even when all of them are down they will eventually get up. Clearly the requests won't go through when all the backend servers are down and the healing works as long as the proxy server is up.

In the future improvements of the data recovery the following script could be easily added considering current functionality:

If one of the servers is down completely for some amount of time we could have a functionality that will help to introduce a new other server at the exact same position so it takes the responsibilities of the failed one. (now new node introduction doesn’t have this feature) New node introduction is covered in the following section.

Discussing System performance

1. PUT Requests Wait Time

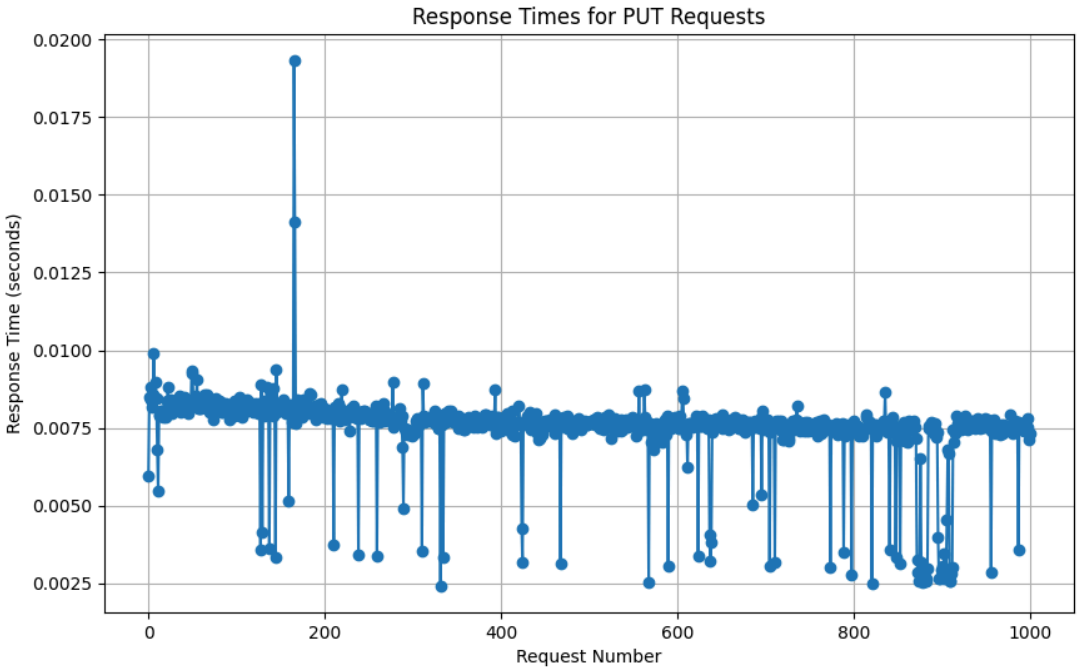


Figure 8. Graph representing the load under 1000 PUT requests

The time series represents the waiting time for the servers. As we can see, the latency is around 7.5 ms. But we do have some outliers on both ends. Since we could have other students interacting with the server this could affect the load of the servers and explain the outliers.

2. GET Responses Wait Time

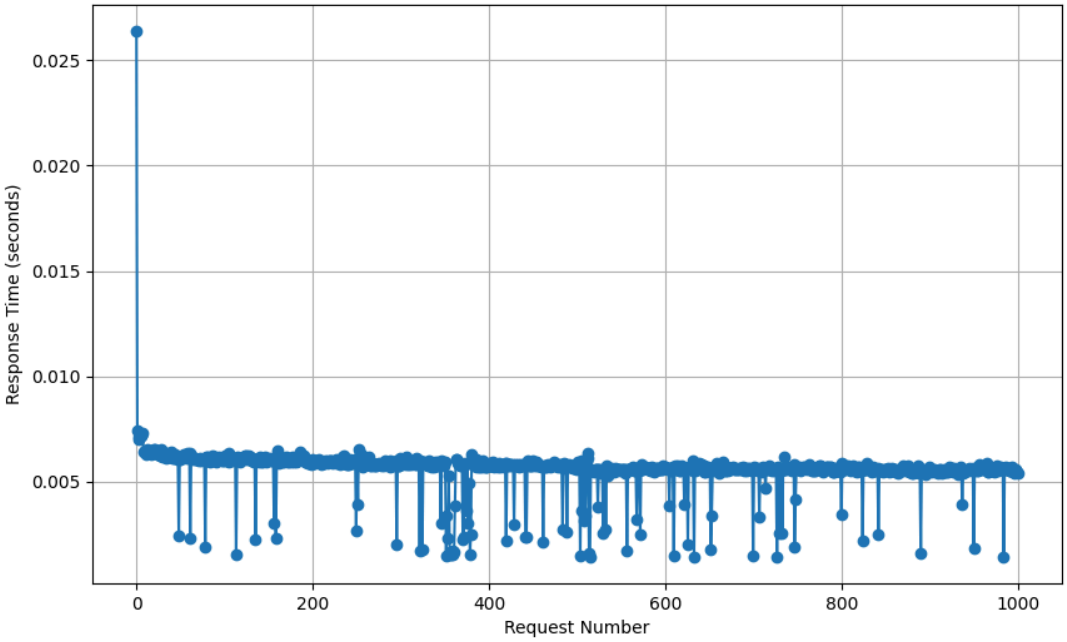


Figure 9. Graph representing the load under 1000 GET requests

The average GET request requires **6ms** but we do have outliers. The outliers that are **below 3ms** are most likely **cache hits**.

By analyzing these graphs we can see that almost all the requests take the same time. That means that out system is quite reliable regarding the response time.

Scalability

1. Horizontal scalability

Introduction of new Node

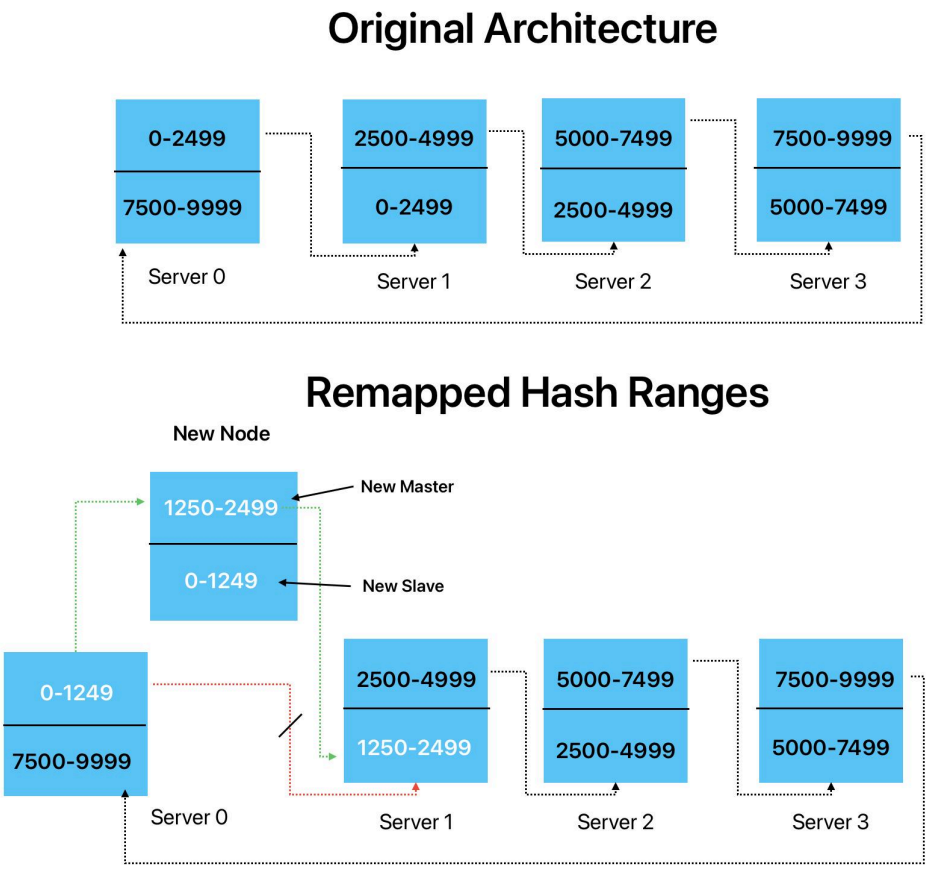


Figure 10. Introduction of new node.

When the system is under high load, a node will be added to share the load.

This diagram shows that although the servers do not take an equal proportion of the load to the number of servers, it means that rehashing can be avoided as the hash ranges are reassigned and data is copied to the new server.

Future Improvements

For the future, a creation of another proxy server not only reduces the chances of the whole system going down, because currently proxy is the only failure point. But also would allow us to receive more requests, forward more requests and have a bigger cache which means less cache misses.

2. Vertical scalability

We have utilized the full power of the current computers, by using all 8 cores and have a thread pool of 8 to maximize core usage. If we had more cores available or hyperthreading enabled when vertically scaling we could increase system performance.

Another way to take advantage of vertical scalability could be to increase ram to increase cache size and increase storage to store more data per server.

Testing Tools Discussion

- **cURL**

We used this command-line tool to test basic GET and PUT requests, ensuring that the proxy and servers correctly handle incoming requests and provide the expected responses.

For example, we would create a PUT request followed by a GET request with the same short to see if we are served the redirect html. We could also test to see which servers receive the request and how the system functions if a server is added or removed, by using this command along with the proxy server's stdout. This tool was essential for debugging the system and understanding how the data was being routed.

- **Apache Bench (ab)**

We used the benchmark in a1/performanceTesting/LoadTest to test the effectiveness of caching and threading by having tests with the cache enabled or disabled and thread counts varied. The script runs the same GET request a number of times, so we can see how effective our system is at caching the result and returning a reply without creating connections with the backend servers.

This tool highlighted the performance differences in response times and throughput so we could come up with an optimal thread count for the proxy and sqlite servers, as well as optimizing the caching to handle data concurrently.

- **Load Testing Tools**

Similarly to the Apache Bench, we used this benchmark to optimize our final system. We needed to make sure under high traffic scenarios, the PUT requests are still reaching the database and the GET requests are finding the correct data.

- **Orchestration File**

We used the orchestration file to have a simple way of simulating server failures, adding new nodes and redeploying the servers whenever an error occurred. This allowed for quicker debugging as we could quickly kill all processes and restart the servers. In addition, we could see if the system remains stable when a server has been terminated or a new server has been added to the cluster.