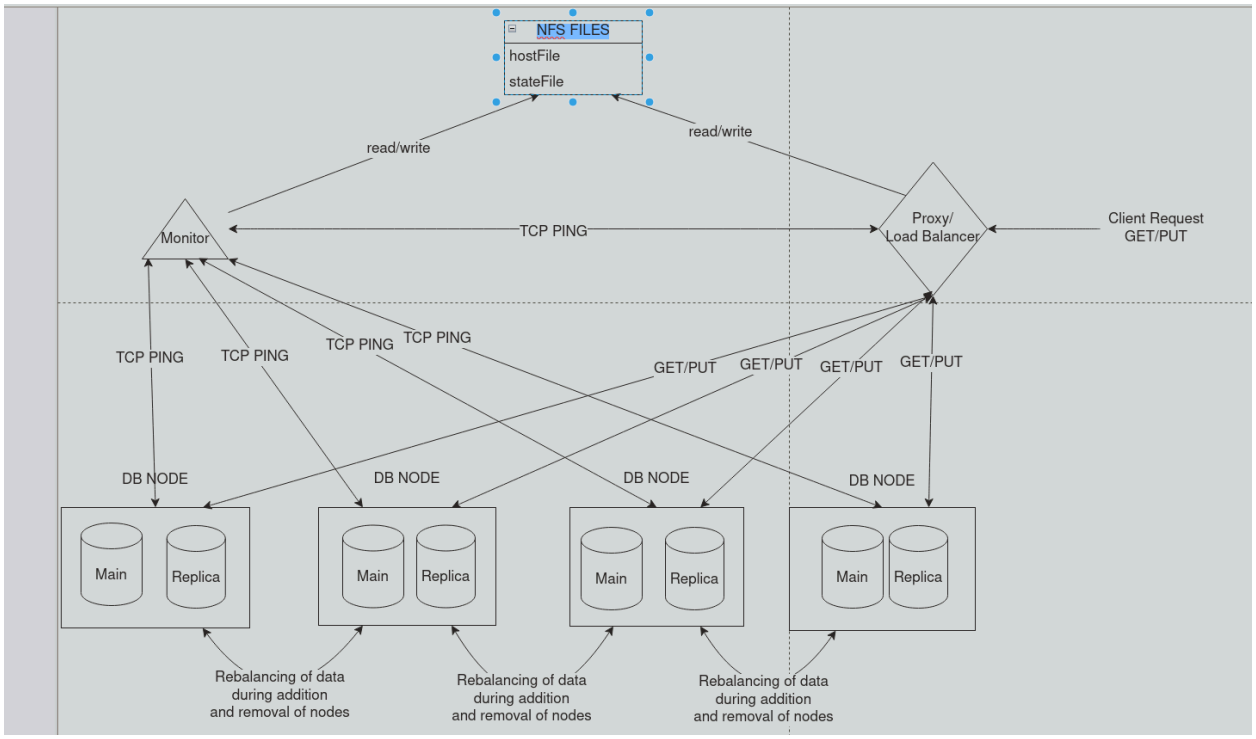
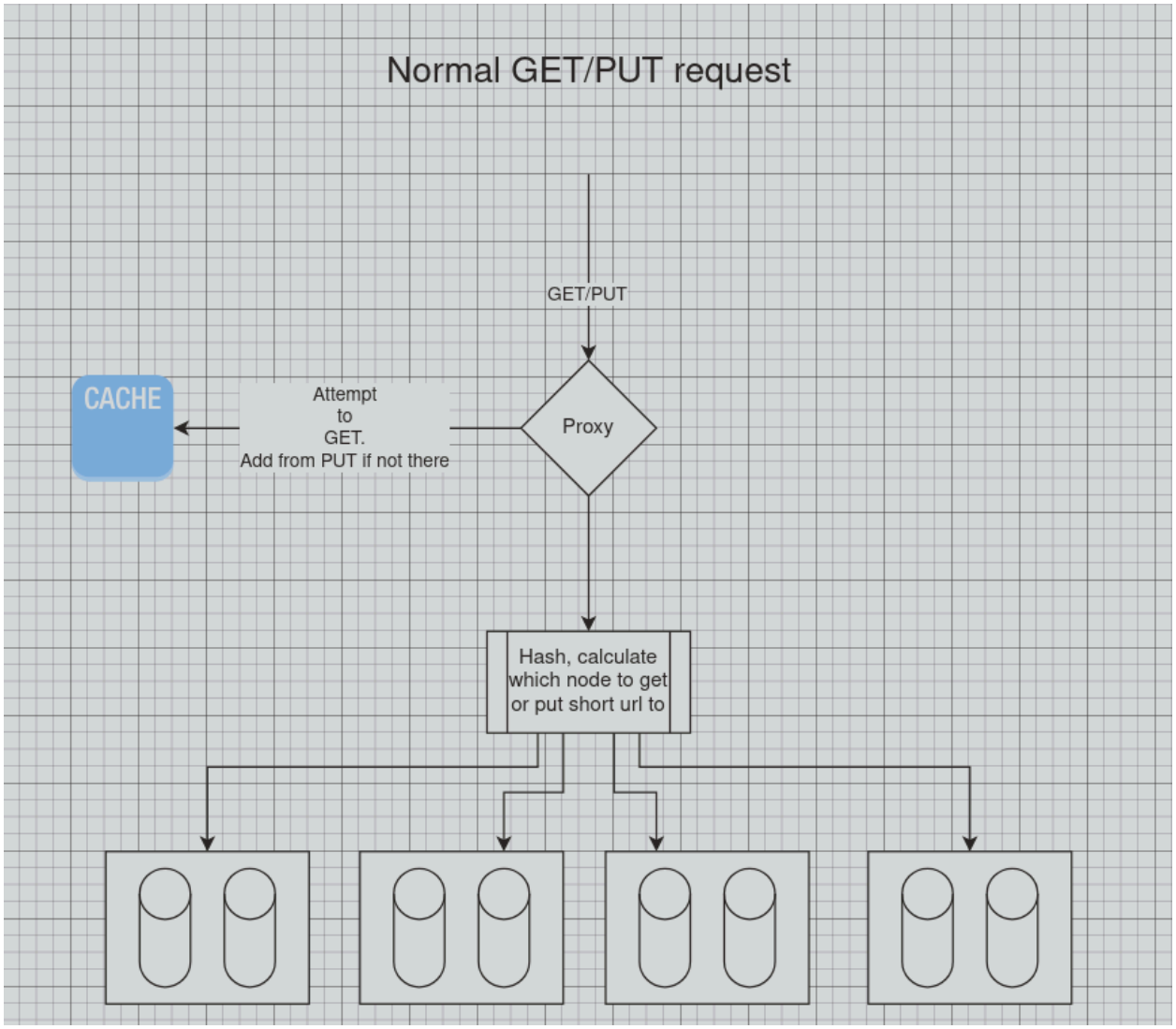
Architecture

Our design looks something like this:

Complete Application Arch and Relationships:



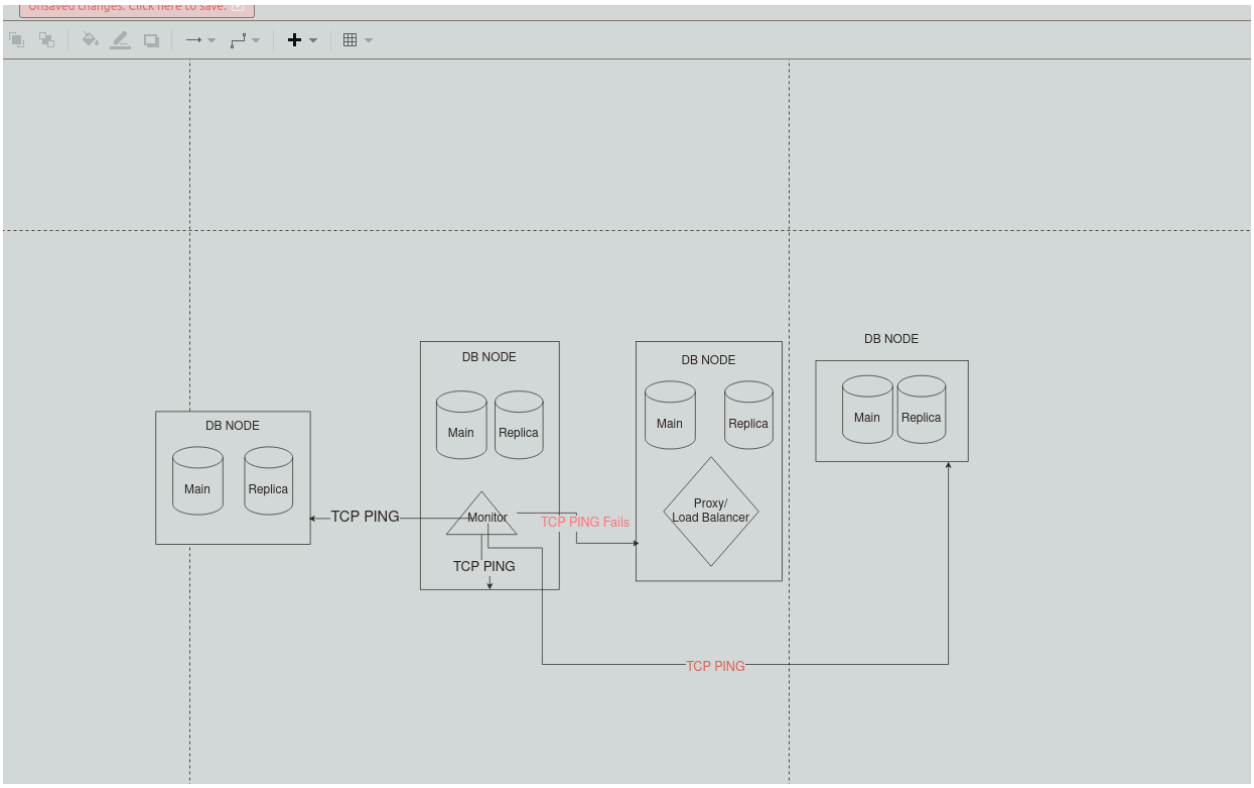
A GET/PUT Request:

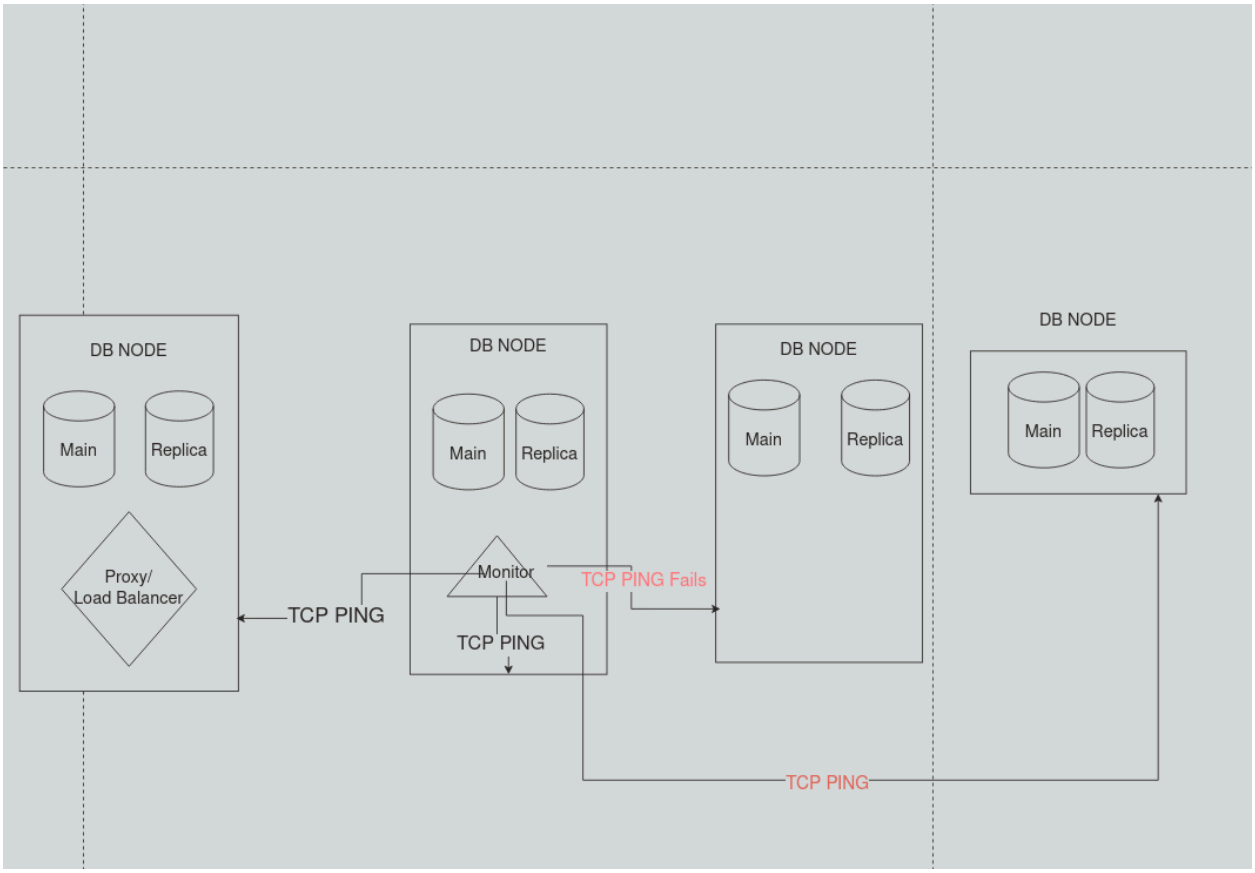


We handle a get request by first checking the cache of the application which is located on the proxy. If there is such a value, we simply return it, without sending another request to the database nodes and recalculating the hashes. Otherwise, we try to query the databases by hashing the shorturl and looking up the appropriate database.

We handle put requests quite similarly, however, we must go beyond the cache for put requests. Even if there is a hit on the cache for that shorturl, we must update this value on the databases as well.

Process of Proxy/Monitor Recovery



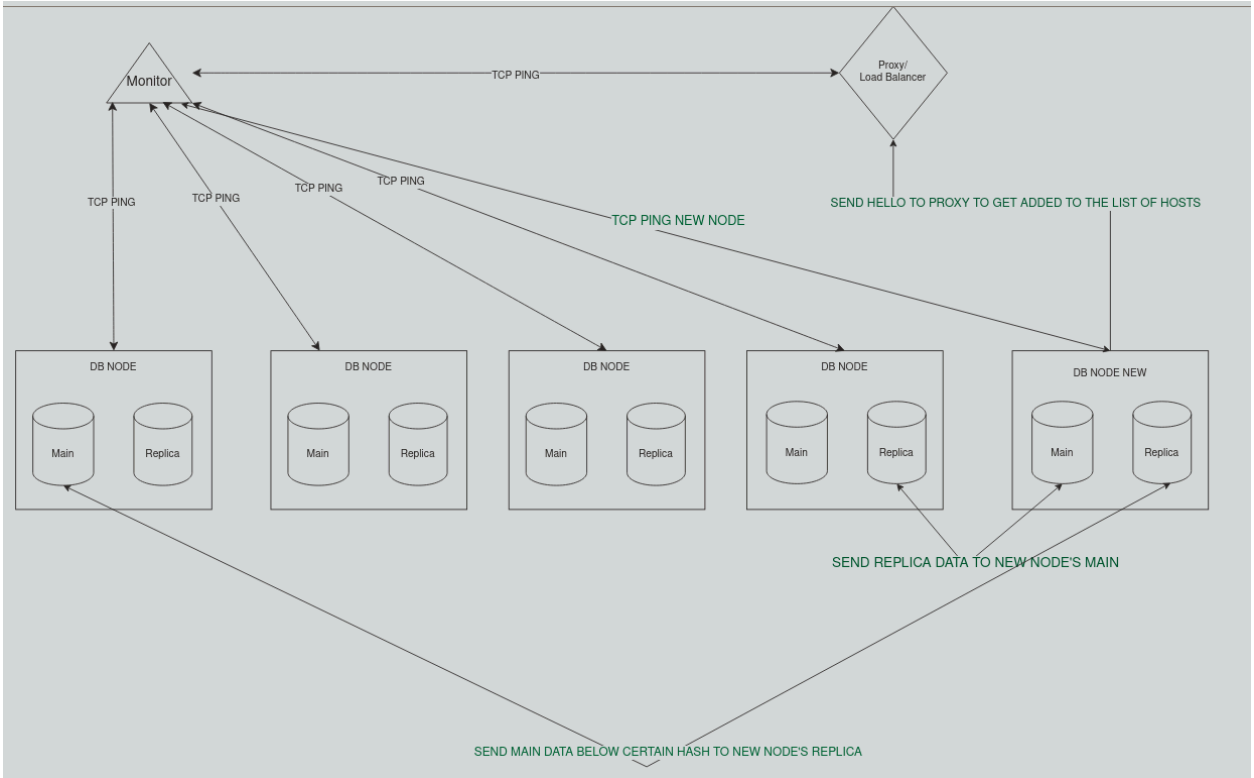


Our architecture is designed so that the monitor and proxy are not located on the same host, as much as possible, anything short of only one host existing. In the case that monitor or the proxy host goes down, we attempt to recover and relocate them on a different host. The monitor attempts a tcp connection to the proxy and does a healthcheck every 10 seconds. It also monitors the db nodes in the same way. If the proxy fails, the monitor attempts to recover it by running a bash script (startProxyRemote.bash), which restarts the proxy on another available node.

Proxy also has a monitor functionality incorporated to monitor the monitor(mouthful, we know). This way if the monitor host goes down, which we would know when the periodic tcp connection attemp fails, we attempt to recover the monitor on a different host.

The system is not foolproof however, if the monitor and the proxy goes down, the service can just be stuck in that state without the ability to recover either of those.

Addition of New Nodes:



When a new node gets added, it sends a hello request to the proxy, which adds the host to the list of available hosts. In addition, this host gets added to the monitor list as well, meaning it will get periodic tcp connection attempts from that point on. When a new node gets added, it also gets added to the circular hash ring logic, and replica data from the node that comes after it gets

added to its main database, while part of the main database from the node that comes before it gets added to its replica database.This way, data gets slightly more balanced and the databases have less data they need to handle on average.

Client Request Flow

- **Client Interaction**
    a. **Client Request (HTTP PUT/GET)**: The system accepts both HTTP `PUT` (to shorten URLs) and `GET` (to retrieve long URLs) requests from clients.
        i. **PUT** request: `PUT /?short={shortURL}&long={longURL}`
        ii. **GET** request: `GET /{shortURL}`
- Client Sends a request
- The Proxy hashes that request based on the short URL. Based on the hash, we obtain a slot
    a. The proxy is load balancing by assigning every DBNode a range of slots. So we hash the incoming short, and assign it a slot, the DBnode that covers that slot is responsible for handling that request.
- The proxy sends a request to the DBNode who's responsible, waits for a response, then send the response of the DBnode back to the client.

Proxy Server Overview

The **Proxy Server (PS)** is the entry point for all incoming requests from clients. It routes these requests to the appropriate backend URLShortner servers based on consistent hashing. Below is a detailed explanation of the proxy server architecture and how it handles incoming client requests.

IP address: 142.1.46.25 Port: 8081 Restrictions: Cannot place monitor application and PS on same node.
NOTES: IP address is on 142.1.46.25, we act like this is a floating IP address. So we assume the DNS has multiple IP addresses attached to this one) However the proxy server can move from one node to another and therefore can actually change IPs.

Function of the Proxy Server

- **Entry Point for Client Requests**: The proxy server listens for incoming client requests on a designated port (e.g., `8081`).
- **Load Balancer**: It is a load balancer, determining which backend URLShortner server should handle the request based on **c**onsistent hashing.
- **Fault Tolerance and Replication Handling**: The proxy also helps manage data replication across nodes and ensures fault tolerance by distributing requests between the **main** and **replica** servers.
- **State Management**, the proxy server manages the state of the connected hosts, and their assigned range of slots. The state is constantly saved so that in the event of the proxy server crashing, on revival it can retrieve itself by recovering the state.
    - The state is mentioned as stateFile throughout the report, but on the system is an java object file.
    - The hostFile is another NFS mounted file that is saved between both

Client Request

- The proxy server listens for requests on a port (e.g., `8081`).
- When a request arrives, the proxy server reads the request to determine its type (e.g., **PUT** or **GET**).

Request Parsing

- The request is parsed to extract important parameters:
    - For **PUT** requests: The short URL (`shortURL`), long URL (`longURL`), and `db` parameters.
    - For **GET** requests: The short URL (`shortURL`).

Consistent Hashing to Determine Target Server

- **Consistent Hashing** is used to determine which backend **URLShortner** server (or node) should handle the request.
    - The proxy hashes the `shortURL` and determines the target node (server) that corresponds to that hash on the consistent hash ring.
    - The proxy selects both a **main node** and a **replica node** for each `shortURL` based on this hashing.

- Hashing algo:
  - We create 10000 slots, as such it is the maximum amount of nodes we can support, we believe it to be a reasonable number of nodes.

```
public int hash(String key) {

        int hash = 0;

        int prime = 31;

        for (int i = 0; i < key.length(); i++) {

            hash = prime * hash + key.charAt(i);

        }



        int scaledHash = Math.abs(hash % 10000);

        return scaledHash;

    }
```

- **Forwarding the Request**
  - The proxy server forwards the request to the target **URLShortner** server (node) selected based on the hash.
    - For **PUT** requests: The data (e.g., short URL, long URL, hash, db parameter) is forwarded to the backend server that will handle the request.
    - For **GET** requests: The proxy forwards the request to the main server for fetching data. If the main server is unavailable, the request can be forwarded to the replica server.

Managing Node Additions and Removals

- The proxy server also handles dynamic scaling, where nodes can be added or removed from the system.
    - When a new node is added, consistent hashing ensures that only a small portion of the existing data needs to be migrated to the new node.
    - Similarly, when a node is removed, the proxy uses consistent hashing to redistribute its data to other nodes.

Data replication, Consistent Hashing, Healing Process

- Each node (or server) in the backend URL shortening system is assigned a hash range on a circular hash ring.
- When a client request arrives, the **shortURL** is hashed, and the resulting hash value is used to find the nearest node (server) on the hash ring that will handle the request.
- The proxy server replicates the data by forwarding requests to a server's main database, and also the replica database of the server that comes after it in the hash chain.
- We use the replicas as a way to heal. By retaining data of the next database(in the hash ring) in the replica db, we ensure that when one server goes down, its data is stored in another database node, meaning it can be recovered.
- When a node goes down, because its data was stored in the hash chain's previous node's replica, we can use the data in the replica when a new node is added. The replica data from the previous server migrates to the new server's replica database, and part of the data from the previous server migrates to its own replica database, and the main database of the next newly added node.

Orchestration

serverSqlite/runit.bash - to run urlshortener on the current host
serverSqlite/start_all.bash - to run urlshortener on all hosts
serverSqlite/shutdown_local.bash - to shutdown urlshortener on the current host
serverSqlite/shutdown_all.bash - to shutdown urlshortener across all hosts
proxyServer/runProxyServer.bash - to run the proxy server on the current host
proxyServer/shutdownProxyLocal.bash - to shutdown the proxy server on the current host.
serverSqlite/runOnHost.bash - to run a host's urlshortener database application from any one of the hosts

Scalability:

- Administrative scalability: how easy is it to add a new host:
    - New hosts are quite easy to add, once a new host comes online via the urlshortner app, it sends a request to the proxy letting it know that it is online now. As a result, the proxy sends the appropriate requests to facilitate the data migration and calibrate the hashing for future data.
    -
- Size scalability: Can we scale up space or performance by adding more hosts? After adding a new host, what kind of performance improvement could we expect:
    - We can scale up space by adding more hosts, since each new host gives us more storage to use. Assuming our hashing algorithm is well-balanced Each new node we offer is likely to distribute the load more, and as a result each individual node will have less overhead to perform and as a result perform better.
    - As we scale up, we gain more threads per host! As a result we have a lot more computational power to handle all requests.
- What are the bottlenecks:
    - Bottlenecks are our program's limit on the thread amounts. Additionally, we are locking the database for write operations, which means the threads must wait to write on the databases, and this is serialized.
    - Another bottleneck is the CPU performance and RAM of the node running the proxy server running as it handles all the ongoing connections with both the clients and the Db Nodes. During loads, having more CPU and RAM could result in a better system.

Latency: how long it takes to return a response to a request could be max, average, or percentage.

Throughput: requests/transactions per second

Availability:

System remains available and consistent so long as:

1) 1 of the hosts are up

2) Proxy server is up

3) It might not have all the data when more than 2 db hosts go down, however as long as 1 host is up, the proxy will be relocated to the available host, using the monitor

4) Monitor is also relocated in the case that the host that the monitor is located in goes down. It will spawn in another host and continue operations.

Durability: lifetime of your data and guarantees against failure. Again, you can consider Service Failures, Host Failures, Storage Failures Examples: Storing in RAM: Any host or service failure results in permanent data loss. Partition Data onto different Hosts: If k of n services/host/storage fail, then (n-k)/n of data is still available. Replicating Data: If fewer than 1/3 of services/host/storage fails, all data is still permanently available. If a node fails (service/host), all data written to that node one minute prior to the failure is still permanently available.

Since every host keeps primary data, and a replicated data of another host database, each hosts' data is durable as long as its primary and replicated node is durable. So if for any host, the two of the hosts go down without the data getting replicated, that data is unfortunately lost.

Consistency guarantees:

We can guarantee consistency between the primary and the replica nodes since we only have a replication factor of 2. Since every PUT request that is received by the proxy gets forwarded to two servers, where in one it goes to the main database, and in the other it goes to the replica database, the data will be the same.

This has limitations however. If at the time of the put requests being forwarded, one of the db hosts goes down, this data might be lost at the lost host. The same applies to the proxy going down during this operation.

We will attempt to recover the data through the hostFile and stateFile put to the NFS server by the host pcs, however this has limitations

Caching:

Caching is implemented through a LinkedHashMap data structure, which is a hashmap that also tracks the least recently added and most recently added data to the map. This is used to implement the LRU logic, meaning the least recently used item is the one that will get evicted from the map if cache fills up.

We initially intended to implement a cache on the proxy server, however after prioritizing the functionality of the proxy server, we realized the addition of a cache functionality to our proxy increased the complexity of our code beyond a threshold that allows for debugging. So we opted to keep our cache on the db nodes.

Horizontal scalability:

We gain horizontal scalability through the addition of more hosts, since each new host adds more power, more ram, more storage.

Vertical scalability:

Increasing the RAM and CPU, will help with the performance of the proxy server, which in turn could increase the performance of the system. The CPU speed could also benefit the applications running the URLShortener applications as it could help speed operations.

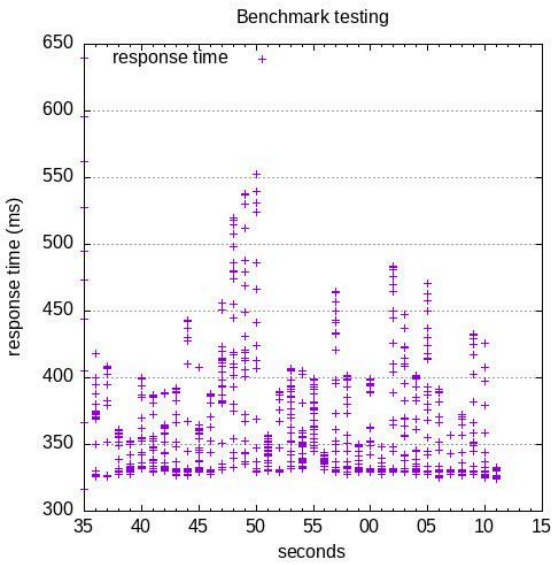
Increasing storage would mainly benefit the database applications, as they would be able to store a larger volume of storage. Increasing the type of storage may be more beneficial as the size of the database is relatively small and hence a faster storage device (magnetic disks> solid state > nvme drive), may be a better for upgrading storage.

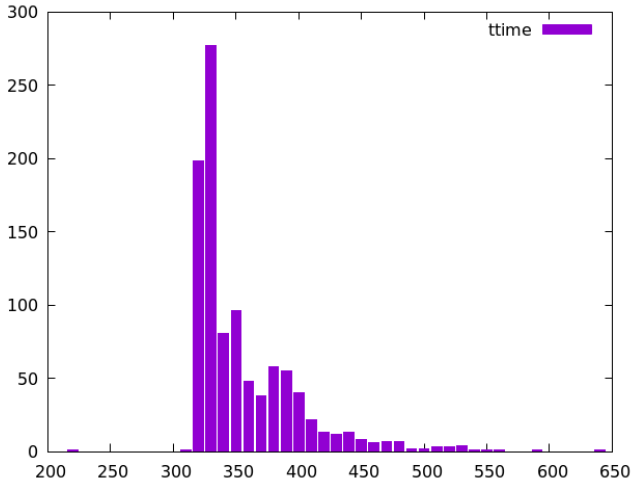
Load Balancing:

The proxy server serves as a load balancer and it forwards the requests to the database servers in the same fashion that the data is replicated by the hashes. Whichever bucket the shorturl hash falls to, the get or put request is directed to that database.

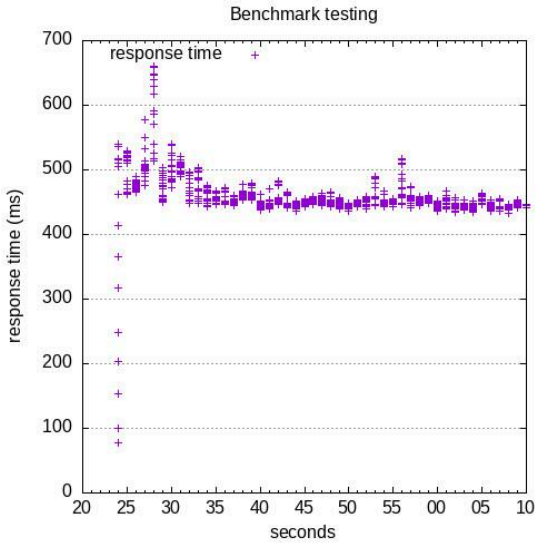
System Performance:

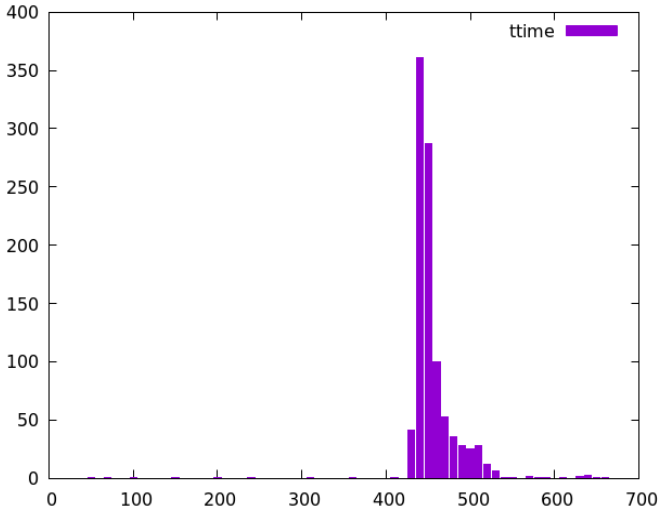
LoadTest1:





Load Test 2:





We can see that most requests for LoadTest1 are completed within 300-350 ms, which represent all the put requests. Some requests will take closer to 650ms, but no more than that, and some requests may take between 200-250 ms, but no less.

Additionally we can see that for LoadTest2 most requests are completed within 450-500 ms, which represent all the get requests. Some requests take between 650-700 ms to get a response, but no more than that.

Not exactly sure why but the get requests are slower than put requests. This might imply that retrieval of data from our database is slower than writing to our databases, however most implementation decisions (creating an index for getting data from the db, synchronization only impacting writing to our cache) intuitively suggest get requests should be faster, however, they are not.