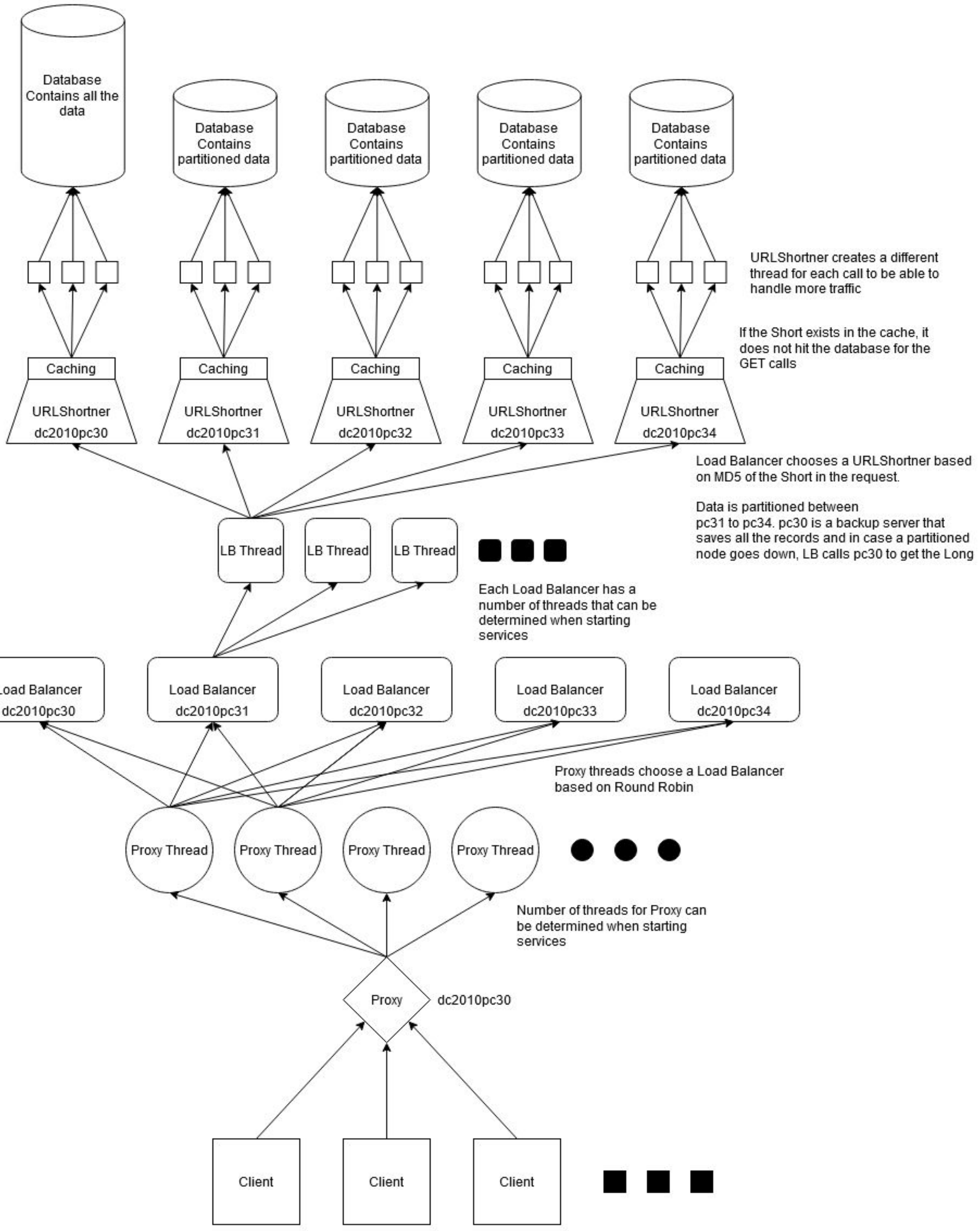
Assignment 1 - group77 - sharifp6, sahgalro, maguang

Getting Started: run `python3 startService.py`

System Architecture:

Below is a diagram laying out the architecture of our URL Shortener service.


Database Contains all the data
Database Contains partitioned data
Database Contains partitioned data
Database Contains partitioned data
Database Contains partitioned data
URLShortner creates a different thread for each call to be able to handle more traffic
If the Short exists in the cache, it does not hit the database for the GET calls
Caching
URLShortner dc2010pc30
URLShortner dc2010pc31
URLShortner dc2010pc32
URLShortner dc2010pc33
URLShortner dc2010pc34
Load Balancer chooses a URLShortner based on MD5 of the Short in the request.
Data is partitioned between pc31 to pc34. pc30 is a backup server that saves all the records and in case a partitioned node goes down, LB calls pc30 to get the Long
LB Thread
Each Load Balancer has a number of threads that can be determined when starting services
Load Balancer dc2010pc30
Load Balancer dc2010pc31
Load Balancer dc2010pc32
Load Balancer dc2010pc33
Load Balancer dc2010pc34
Proxy threads choose a Load Balancer based on Round Robin
Proxy Thread
Number of threads for Proxy can be determined when starting services
Proxy
dc2010pc30
Client

The proxy:

The proxy is running on one host, it can have multiple threads. Having multiple threads helps processing multiple calls from different clients at the same time faster as the clients do not have to wait until the previous call is finished if a thread is not busy.

Load Balancer:

We have a load balancer on each host. Each load balancer also can have multiple threads. This way we can divide the work for processing the calls between different nodes and help improve the performance. We are using MD5 hashing of the Short to distribute traffic to different URLShortner hosts. Based on our testing, for 100000 calls, each host gets around 25000 calls which shows that our traffic distribution based on MD5 works properly.
We partition our data on 4 nodes (Can be changed). There is also an extra node for backup that stores all the data. If a partitioned node goes down, the backup node is used to store or retrieve data which helps with the partition tolerance of our system.
Everytime there is a PUT call, both the partitioned node (Which is chosen based on the MD5 hashing of the Short) and the backup node are called and store the data. When there is a GET call, only the partitioned node is called, however if the partitioned node is down, the backup node is called.

URLShortner:

We have a URLShortner running on each host. It creates a new thread for each node. Each URLShortner also caches the data the host records. We use LRU for caching.

Database:

We have a sqlite3 database on each host. There is one host as a backup that stores all the data and the rest of the nodes are partitioned.

Orchestration:

We have a Python script that opens a user interface for interacting with our program. The orchestration script/entry point of our system is ran using **`python3 startService.py`** can be run with the following commands:

- **Start**: Kills running servers and deletes old database/log files for fresh instance. Starts the proxy, load balancers, URL Shorteners, and creates new log/database files. Also starts monitor in background.

- **Monitor**: View the output produced by the monitoring script running in the background.

- **Stop**: Kills running servers. Keeps old database/log files for reference upon shutdown.

- **Status**: Shows the current status of the proxy, load balancers, and URL Shorteners.

- **Exit**: Exits the user interface.

The monitor (automatically started) after the **`start`** is called, runs in the background and observes all the services running. If one of them goes down, it restarts it.

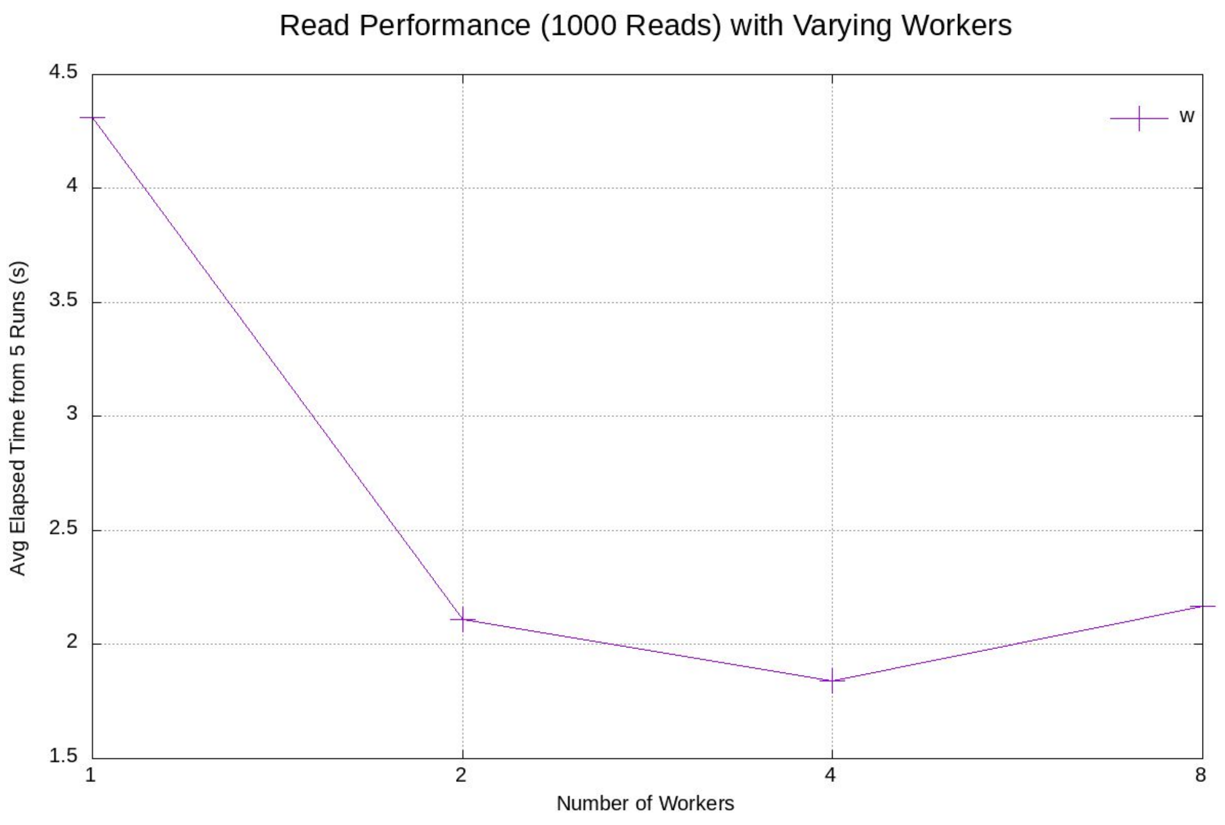
We have a **config** file that contains information about what ports to use, what the cache size should be, as well as which hosts are to be used. Formatted as follows:

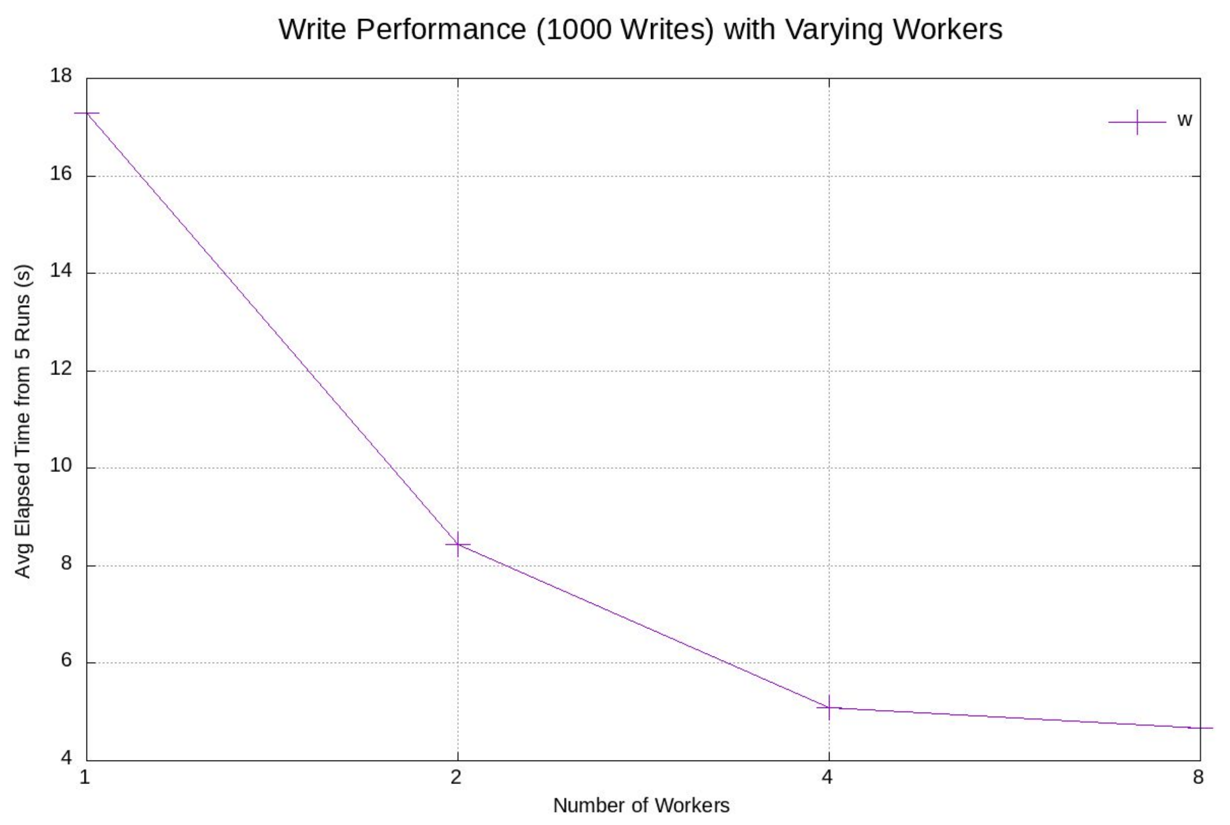
```
proxy_port load_balancer_port url_shortener_port
cache size
host_1 hash_range_start hash_range_stop
host 2 hash_range_start hash_range_stop
…
```

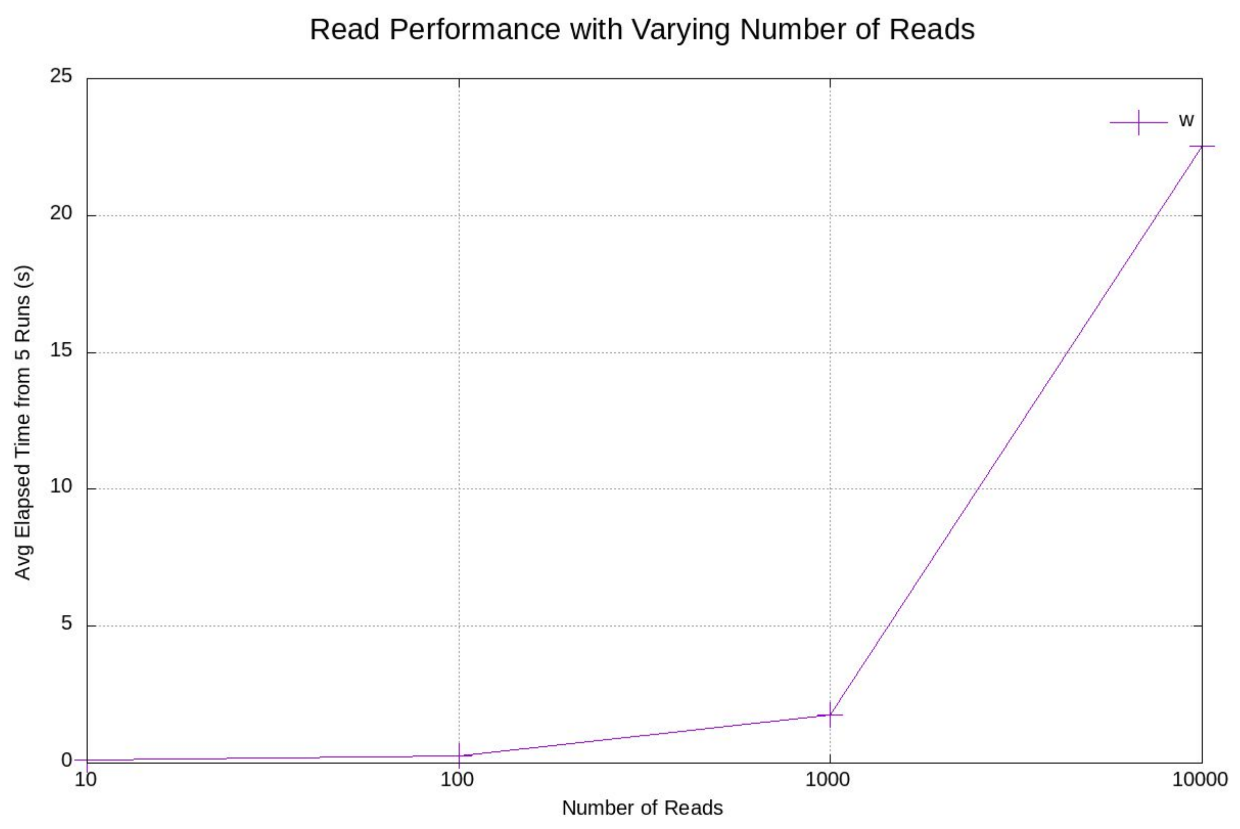
The hash range is the range of hashes each host handles when the requests are hashed. There is always a backup node that covers all possible hashes (0 - 255 in our case).

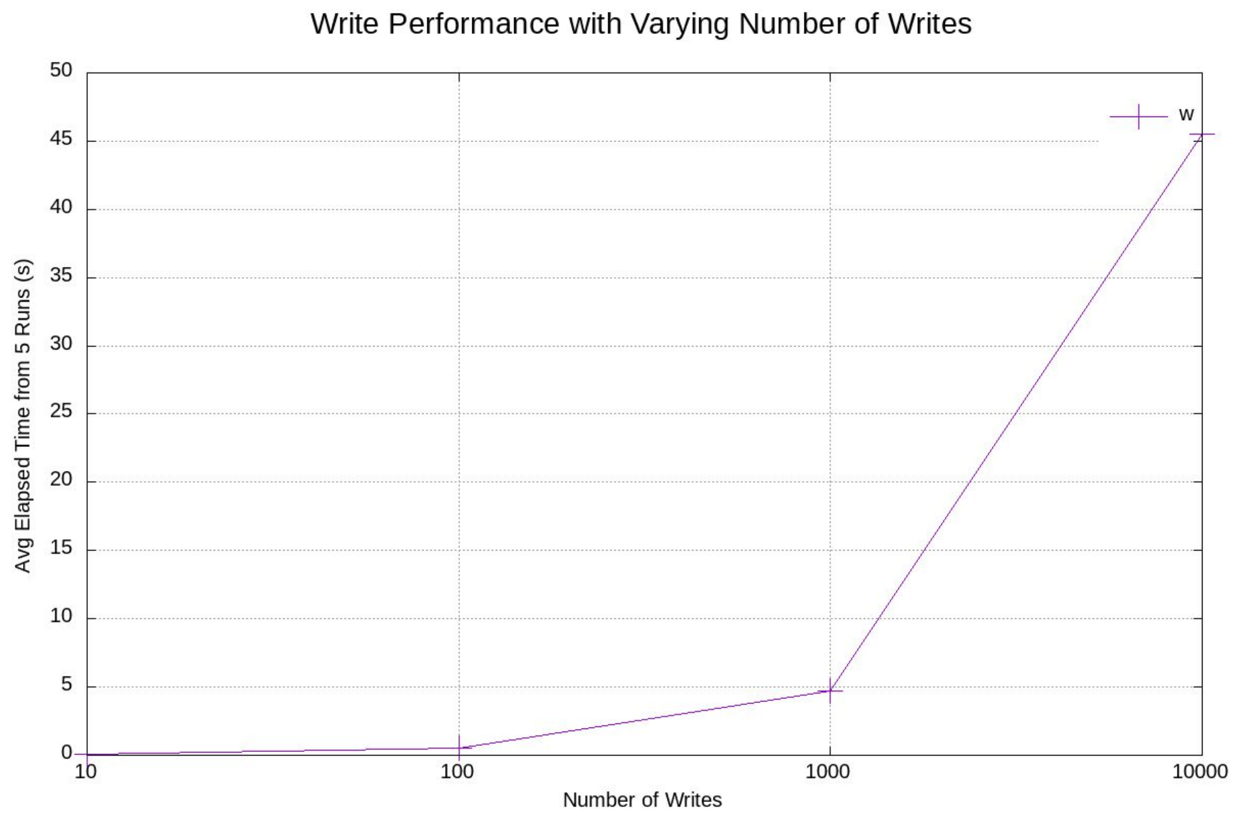
Testing:

Both performance and availability were tested through 4 scripts under `/performanceTesting` and the corresponding graphs can be shown below. For performance, both read and writes with the system were tested with varying workers (performance) and varying reads/writes (availability). These graphs can also be found under `performanceTesting/graphs`.




Write Performance (1000 Writes) with Varying Workers
18
16
14
12
10
8
6
4
1
2
4
8
Avg Elapsed Time from 5 Runs (s)
Number of Workers
w


Read Performance with Varying Number of Reads
25
20
15
10
5
0
10
100
1000
10000
Avg Elapsed Time from 5 Runs (s)
Number of Reads
w



Strengths:

Partition Tolerance
Multithreading in each system that increases the performance
Vertical Scaling
Horizontal Scaling (Scaling up CPU, RAM and Storage in a node improves performance)
LRU caching
Concurrency
Availability

Weaknesses:

Data Recovery
Backup node might have issues with storage
Dynamically adding new nodes