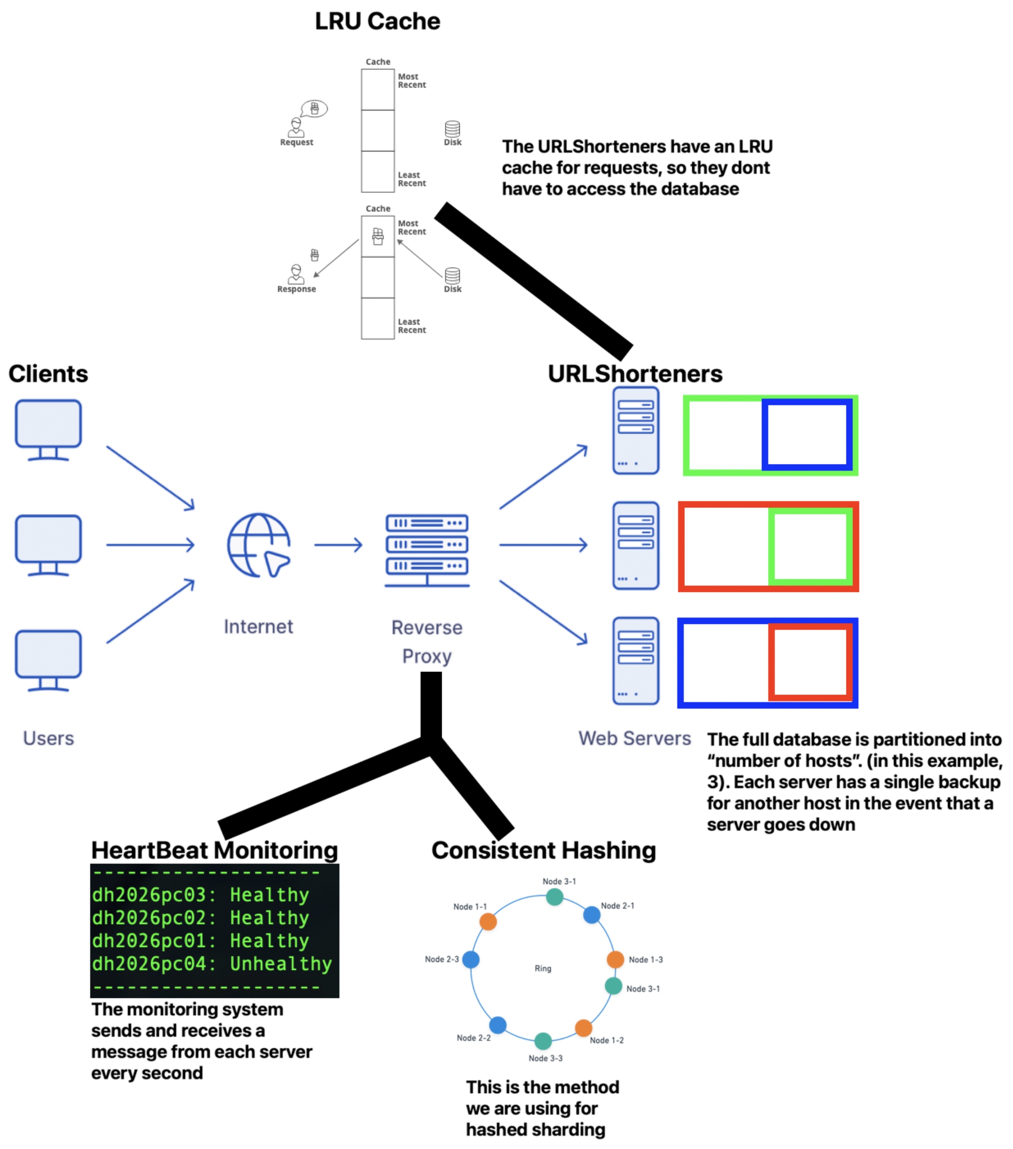
<u>Total weight: 27 points</u>

CSC409 A1 Report

Discussing the system's architecture (17 pts)

- Diagram showing(3 pts):
  - Application system
  - Monitoring system
  - Data flow



The URLShorteners have an LRU cache for requests, so they dont have to access the database

The full database is partitioned into "number of hosts". (in this example, 3). Each server has a single backup for another host in the event that a server goes down

```
--------------------
dh2026pc03: Healthy
dh2026pc02: Healthy
dh2026pc01: Healthy
dh2026pc04: Unhealthy
--------------------
```

The monitoring system sends and receives a message from each server every second

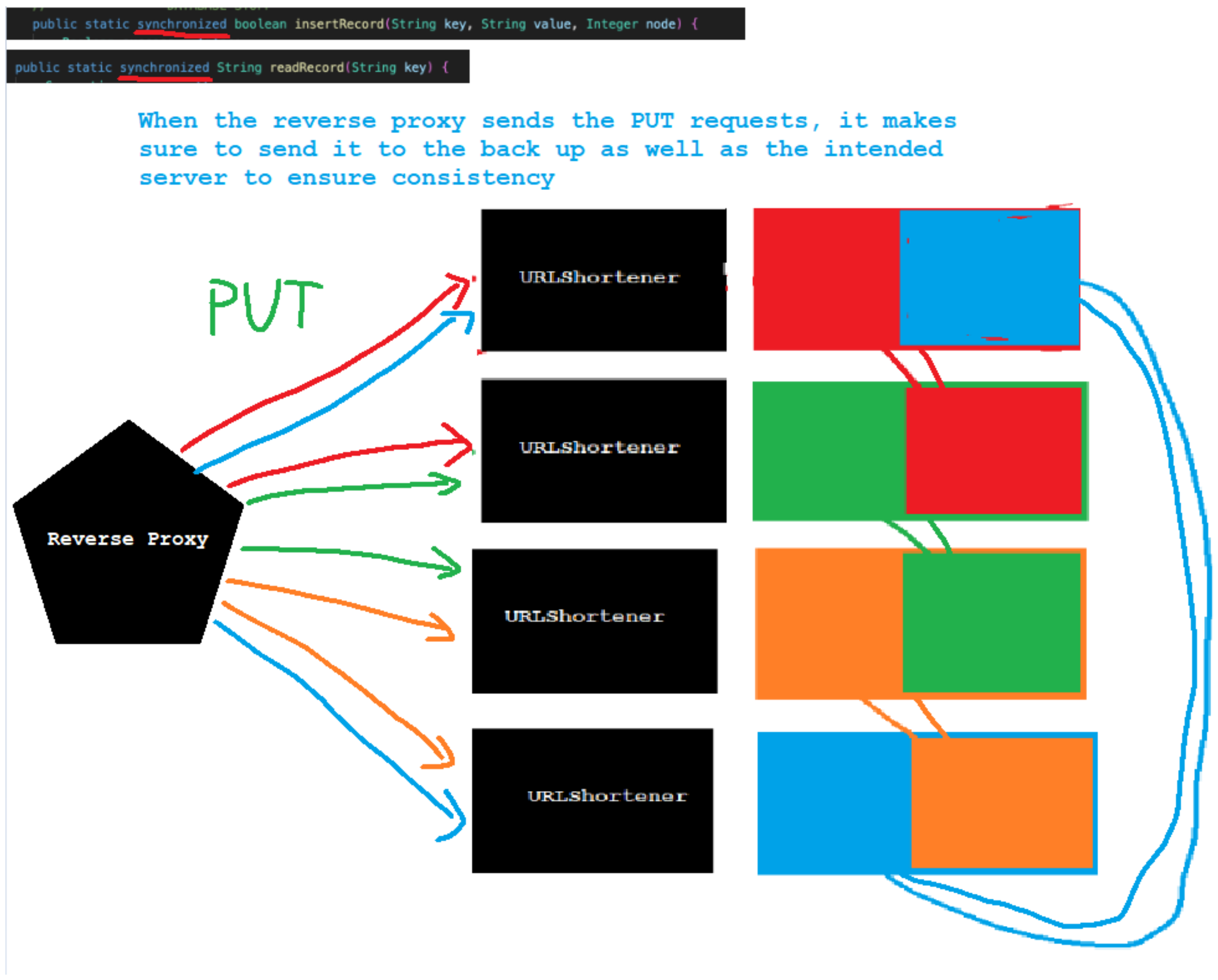
This is the method we are using for hashed sharding

<u>Total weight: 27 points</u>

Discussion of each of the following with respect to your system. For each point, as appropriate, show an appropriate diagram, list performance guarantees, discuss code/architecture choices (14 pts)

- Consistency:
    - Although most of our programs are multithreaded, we have made sure that the cache and the database are synchronized with both reads and writes. This is because we have made the insertRecord and readRecord functions synchronized. Additionally, we are using consistent hashing to determine which database to store the data in and which database to read the data from. This ensures that the data is consistent across all of the databases, as the same short URLs always hash to the same databases. Additionally, we have added fault tolerance in case of a server being temporarily down, which means that the data will be consistent across all of the databases, even if one of the servers is down and comes back up later. Finally, we also ensure consistency by sending PUT requests to both the intended server and its backup, so that the databases always contain the same data. Note that this does actually decrease the throughput for PUT requests, but that is a design decision we decided to make for easier implementation. For the GET requests on the other hand we always only send one, whether it's to the backup of the running server.

Total weight: 27 points

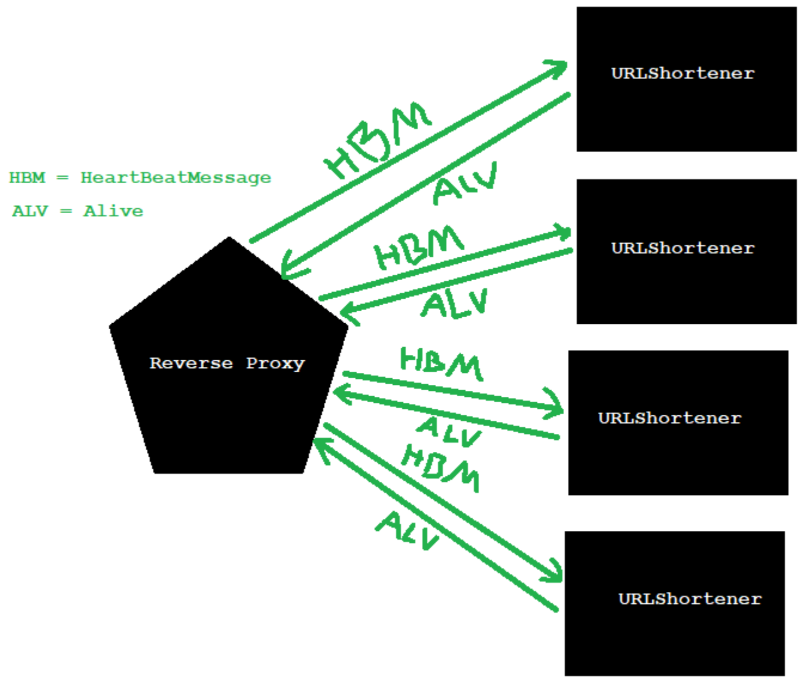
- 

```
public static synchronized boolean insertRecord(String key, String value, Integer node) {
public static synchronized String readRecord(String key) {
```

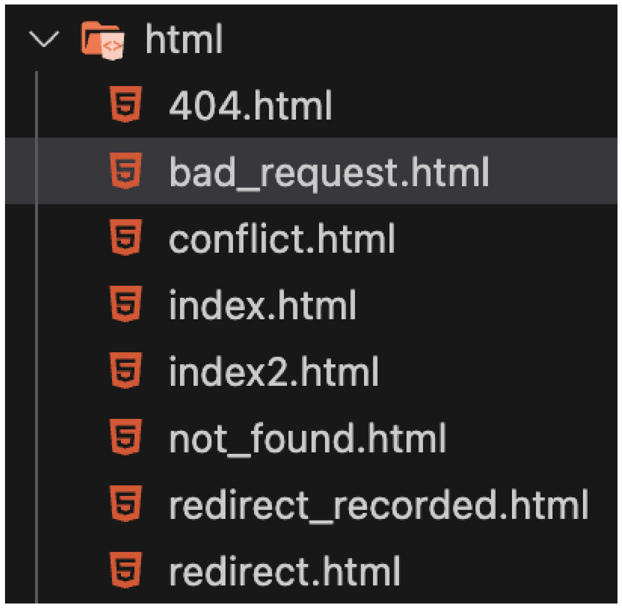
When the reverse proxy sends the PUT requests, it makes sure to send it to the back up as well as the intended server to ensure consistency

- **Availability:**
  - We have implemented heartbeat monitoring to ensure that the servers are up and running. When a server goes down, the reverse proxy is immediately notified, and the requests to the server that is down will be redirected to another server. This ensures that the system is always available, even if one of the servers goes down. We have also ensured that we have error checking to ensure that the system is always available, even if a bad request is made.
  - For the disaster recovery diagram based on availability, please scroll down to the “Process Disaster Recovery” section

Total weight: 27 points



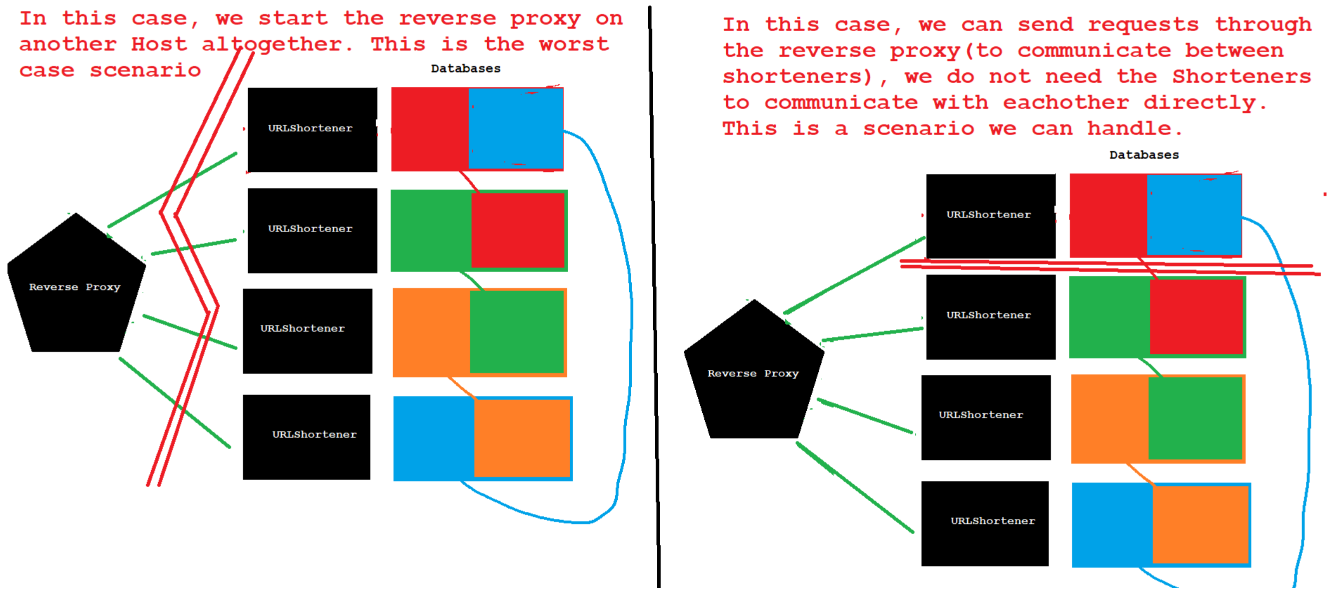
○



html
- 404.html
- bad_request.html
- conflict.html
- index.html
- index2.html
- not_found.html
- redirect_recorded.html
- redirect.html

- **Partitioning tolerance:**
  - If a network partition were to occur between the URL shorteners, the system would still be able to function, as we decided to make most of the necessary communication happen between the reverse proxy and the URL shorteners. However, if a network partition were to occur between the reverse proxy and the client or the reverse proxy and the URL shorteners, the system would not be able to function.

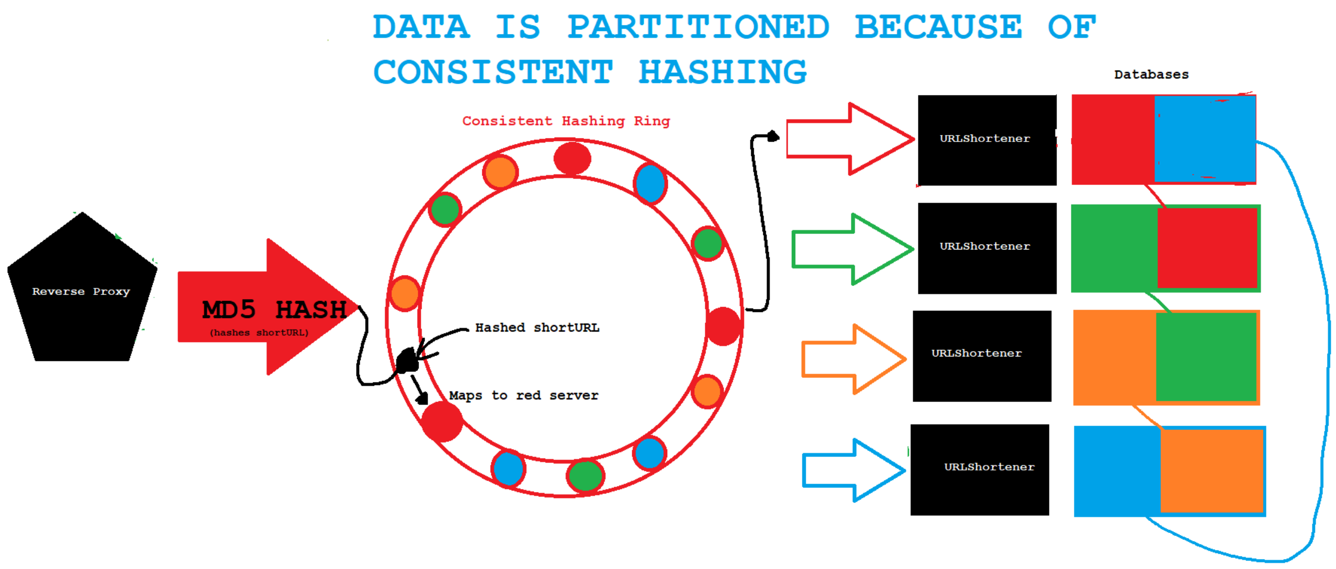
However, all we would have to do is run the reverse proxy on a separate machine.


In this case, we start the reverse proxy on another Host altogether. This is the worst case scenario
Databases
URLShortener
URLShortener
URLShortener
URLShortener
Reverse Proxy
In this case, we can send requests through the reverse proxy(to communicate between shorteners), we do not need the Shorteners to communicate with eachother directly. This is a scenario we can handle.
Databases
URLShortener
URLShortener
URLShortener
URLShortener
Reverse Proxy

-

- **Data partitioning:**
  - We are using hashed sharding to partition the data. In order to do this, we are using a consistent hasher to split the requests as well as partition the data reads and writes. This ensures that the data is partitioned evenly across all of the databases, and that the same short URLs always hash to the same databases. The fact that we are partitioning the data allows us to take a heavier load, as we realized that one of the bottlenecks of our system was the database.

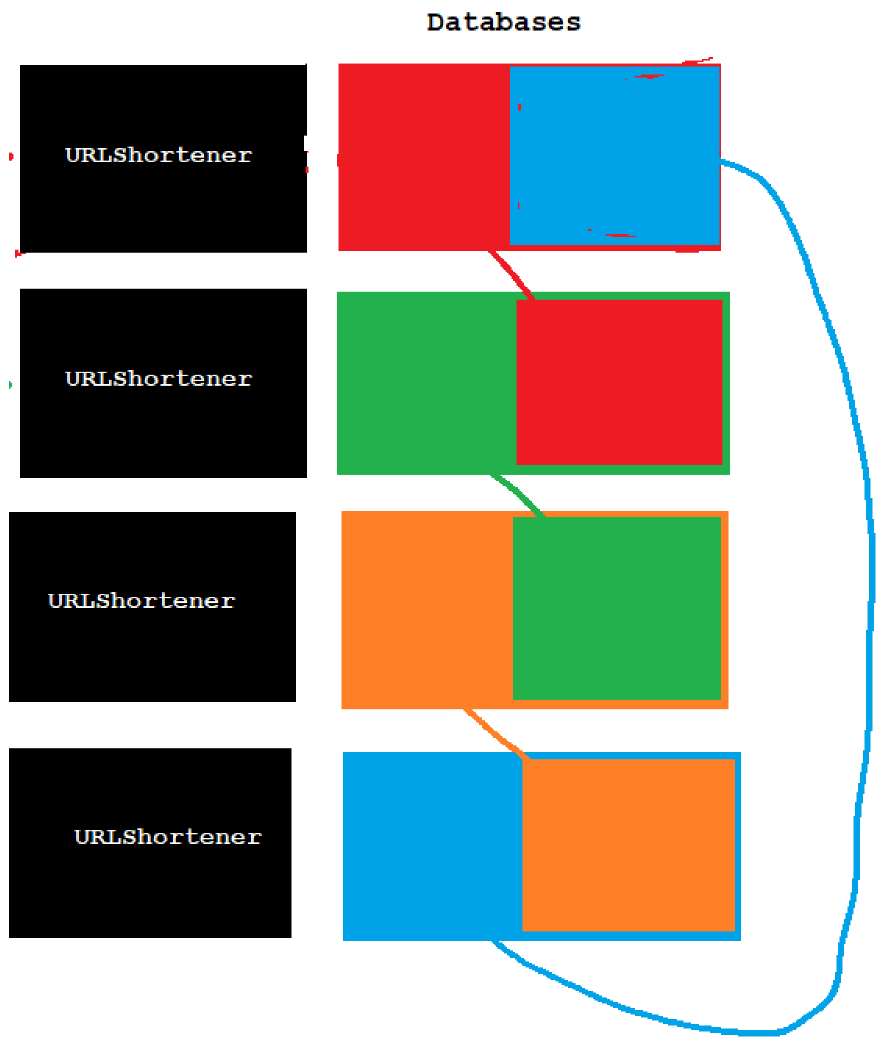
Total weight: 27 points



-

- **Data replication:**
    - Each server has its own database. Each database has a replica of another database. If a server were to go down, the data would be replicated on the server when it comes back up. This ensures that the data is always replicated across all of the servers, and that the data is always available.

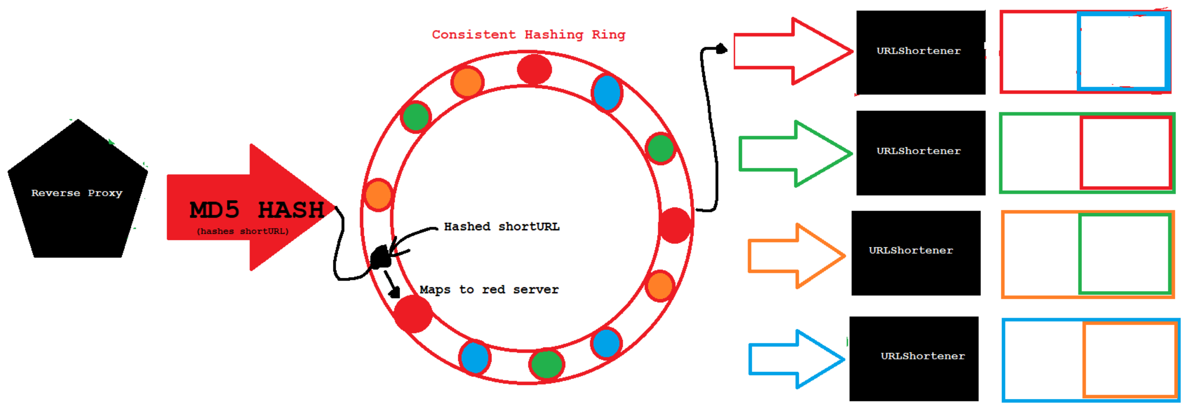
Total weight: 27 points



■

- **Load balancing:**
  - For our load balancing, we are doing our load balancing in the reverse proxy using consistent hashing. Our number of virtual nodes has been optimized based on testing, to get the maximum throughput. The hash algorithm we are using for the consistent hashing is a simple MD5 hash. Although MD5 might be slow by some standards, it is not the bottleneck of our system, and it is fast enough for our purposes.

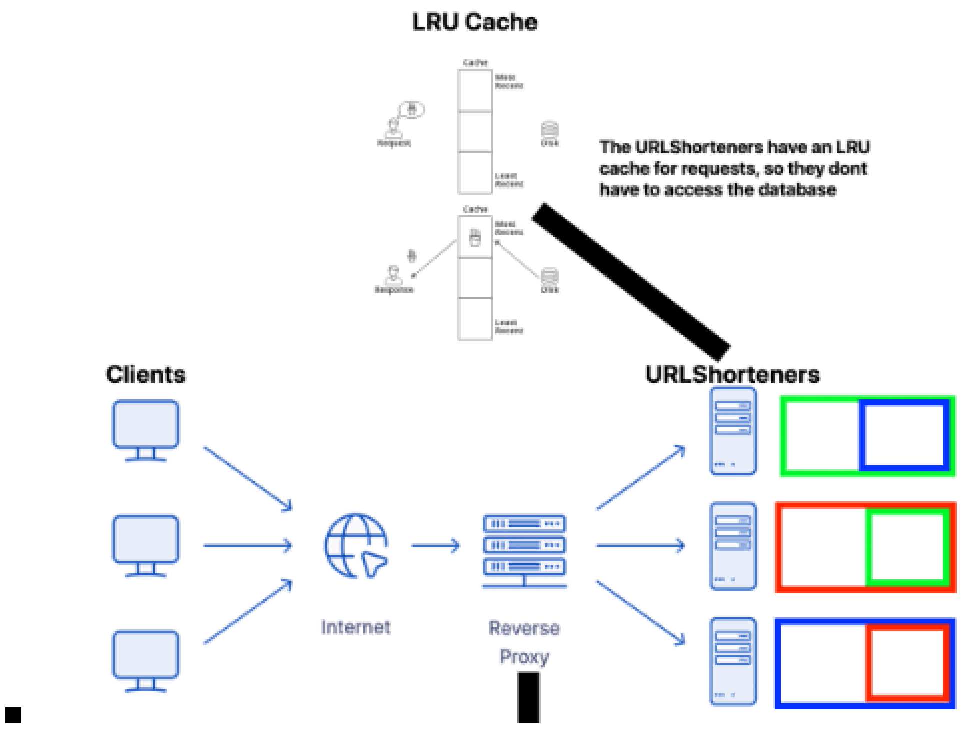
Total weight: 27 points



-

- **Caching:**
    - For the caching, in the URL Shortener, we are using a simple LRU cache. We are using a synchronized hashmap to store the data. The number of cache items has been kind of optimized, we say "kind of", because it would really depend on the frequency of same requests we get, to choose which algorithm to use. We have decided to use a simple LRU cache, as it is simple to implement and has good general performance compared to other more request frequency-specific algorithms. By caching, our system's throughput is much faster, as we do not have to query the database for the same requests over and over again.

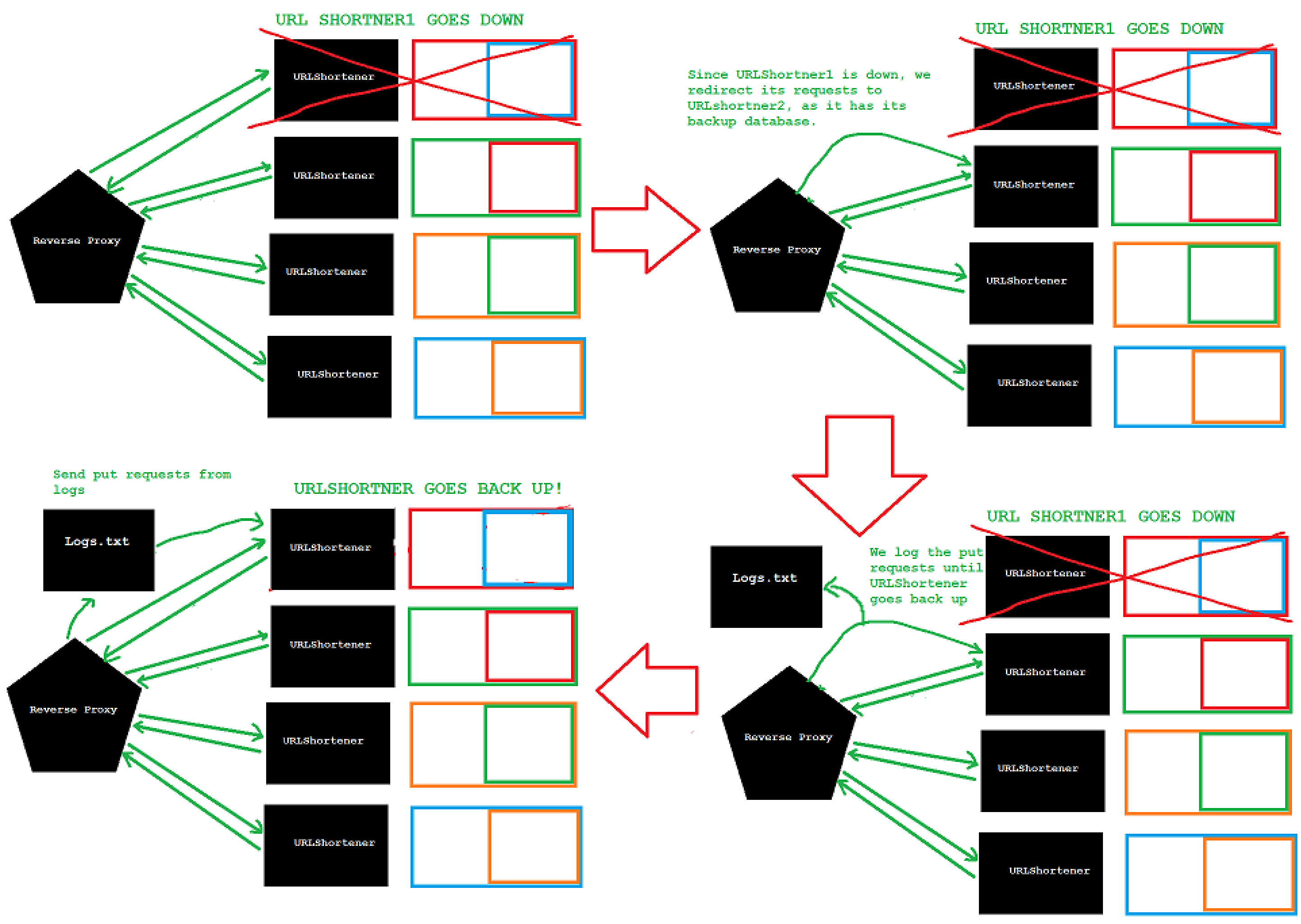
Total weight: 27 points

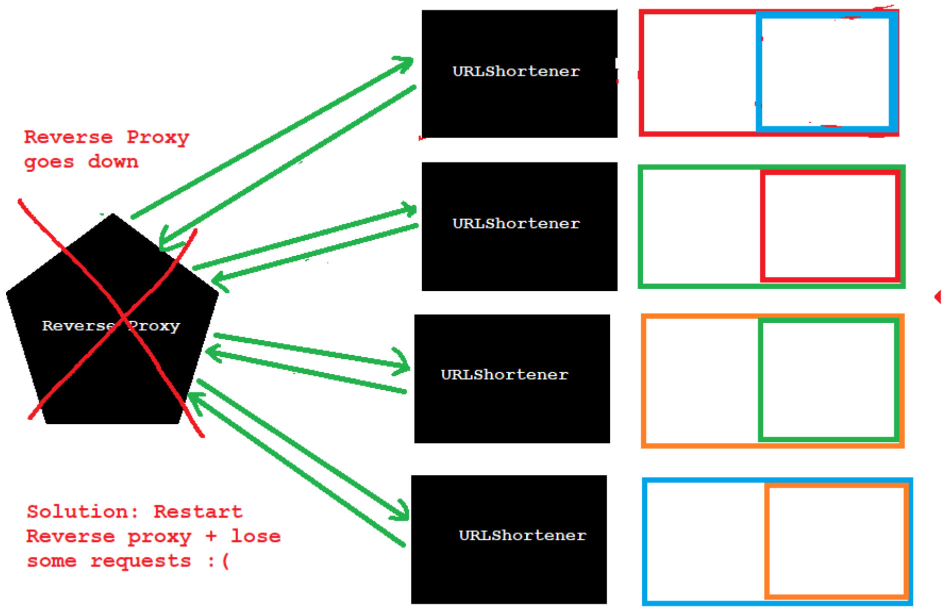


The URLShorteners have an LRU cache for requests, so they dont have to access the database

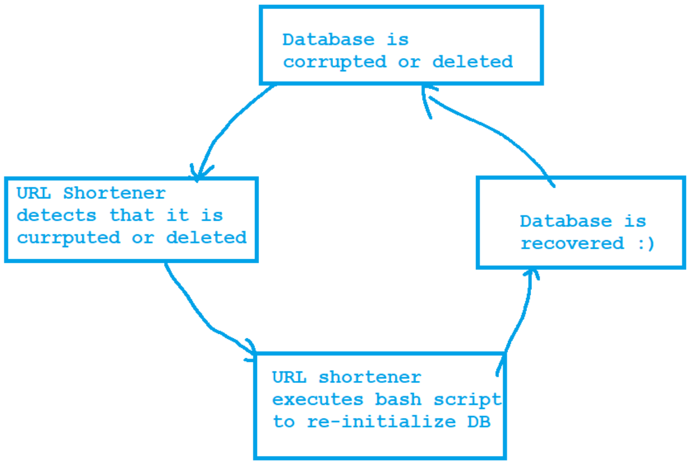
- **Process disaster recovery:**
    - If any URL Shortener process were to go down, the reverse proxy would be notified, and the requests would be redirected to another URL Shortener. Hence, this would not be a problem. If the reverse proxy were to go down, however, the system would not be able to function. We would be forced to run the reverse proxy again on the same machine or on a separate machine. However, this would not be a problem, as the reverse proxy is a very lightweight process. We decided not to make a script for this scenario (to reboot the proxy), as if the reverse proxy were to go down, even for a second, some requests would still be lost, hence there would be no way to avoid downtime.
    - Diagram of URLShortener going down is below and the ReverseProxy going down is below

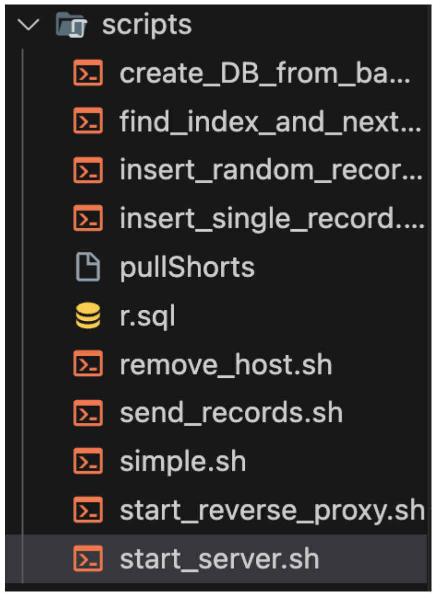
Total weight: 27 points





<u>Total weight: 27 points</u>

- **Data disaster recovery:**
    - The application will execute a script to reinitialize a database of any server if it detects that the database is either corrupt or gets deleted. It will also fill the database with the necessary backup db.
    - 
Database is corrupted or deleted
URL Shortener detects that it is currputed or deleted
URL shortener executes bash script to re-initialize DB
Database is recovered :)

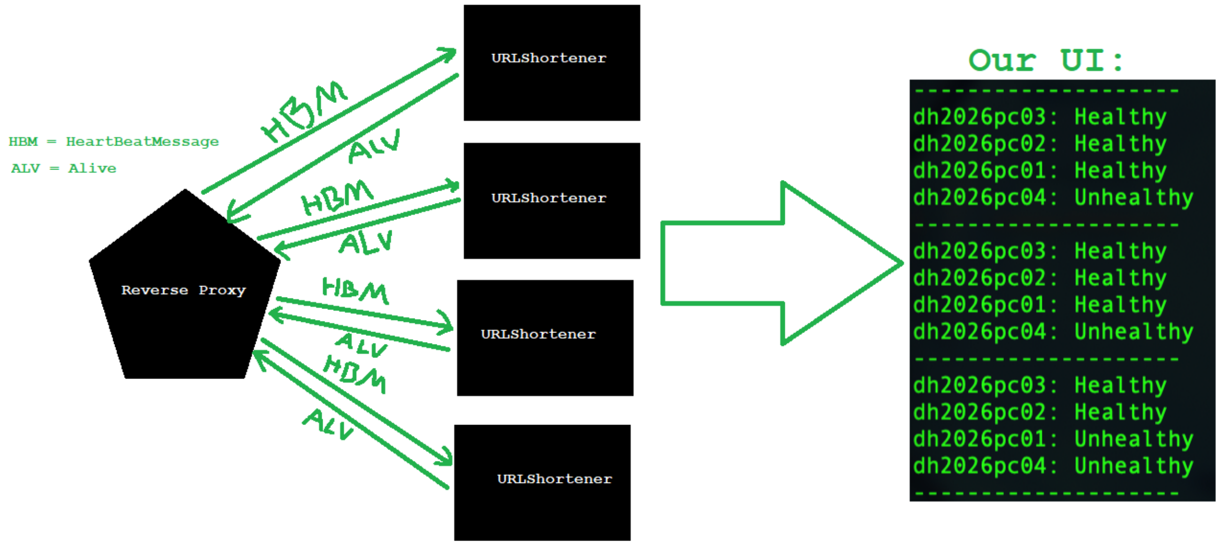
- **Orchestration:**
    - We are using bash script to automate the starting and stopping of the servers (reverse proxy and URLshortner cluster). We also use scripts to add and remove hosts from the host file.
    - 

```
scripts
  create_DB_from_ba...
  find_index_and_next...
  insert_random_recor...
  insert_single_record....
  pullShorts
  r.sql
  remove_host.sh
  send_records.sh
  simple.sh
  start_reverse_proxy.sh
  start_server.sh
```

- **Healthcheck:**

Total weight: 27 points

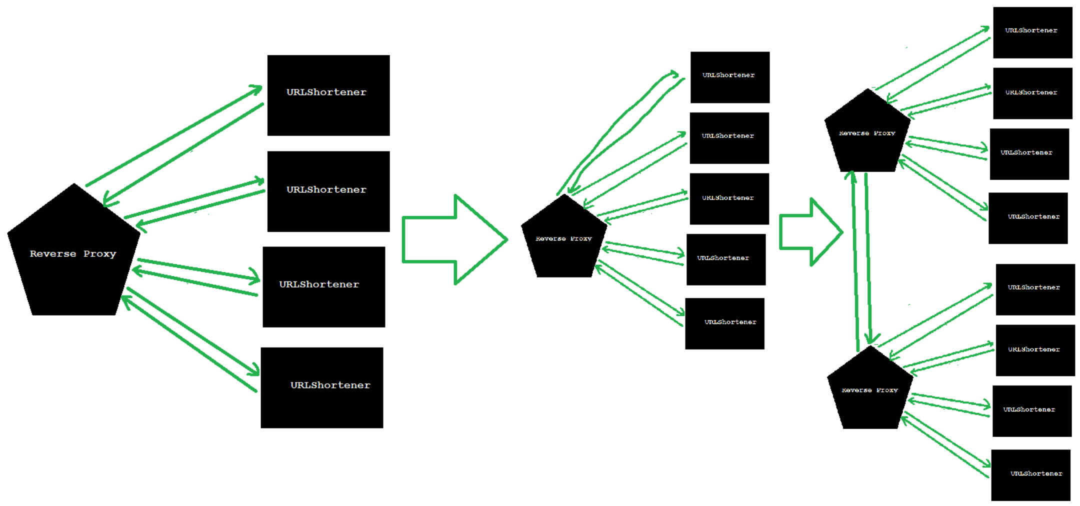
- ○ We are using heartbeat monitoring to check the health of the servers. We are using a simple TCP connection to check the health of the servers. Our reverse proxy simply sends a message to the URL Shorteners, and the URL Shortener responds with an ACK to ensure that it is up and running. We use this health check as our main monitoring system, which we use for other fault tolerance. Note that we chose to do a health check every second, however we could do it a lot faster if need be.



```
--------------------
dh2026pc03: Healthy
dh2026pc02: Healthy
dh2026pc01: Healthy
dh2026pc04: Unhealthy
--------------------
dh2026pc03: Healthy
dh2026pc02: Healthy
dh2026pc01: Healthy
dh2026pc04: Unhealthy
--------------------
dh2026pc03: Healthy
dh2026pc02: Healthy
dh2026pc01: Unhealthy
dh2026pc04: Unhealthy
--------------------
```

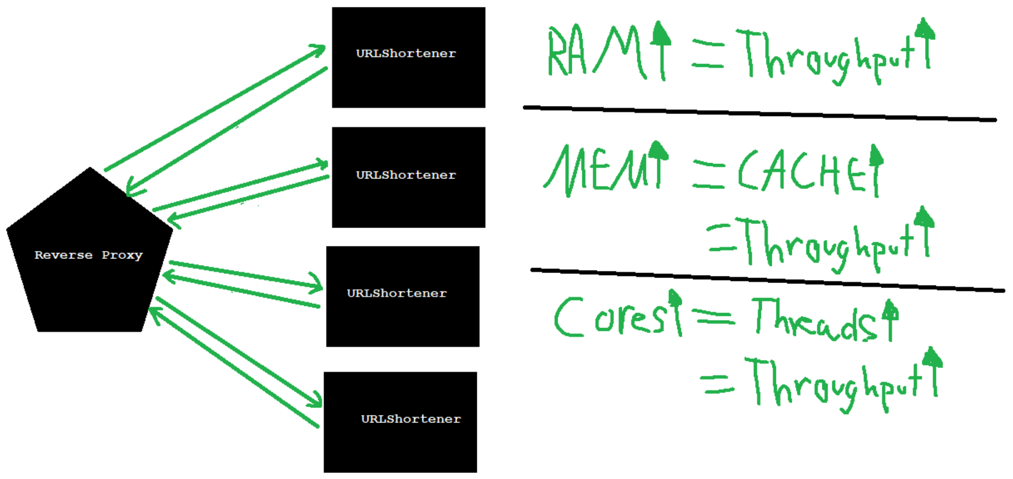
-
- **Horizontal scalability:**
  - ○ By adding more URL Shorteners on more lab machines, we can increase the throughput of our system. After a certain point, we will have to add more reverse proxies so that the reverse proxy does not become the bottleneck of our system.

Total weight: 27 points



■

- **Vertical scalability:**
    - We can increase the throughput of our system by increasing the number of threads in the thread pool. However, after a certain point, too many threads will decrease the throughput, so it is important to find that balance. However, if we were to increase the number of cores in the system or the amount of RAM, we would be able to increase the throughput even further by creating more threads. However, this would not be a good idea, as it would be more expensive, and we would be better off adding more URL Shorteners. This means that our system would indeed be able to take advantage of more cores, more RAM, and more storage (caching) if we were to scale vertically.



○

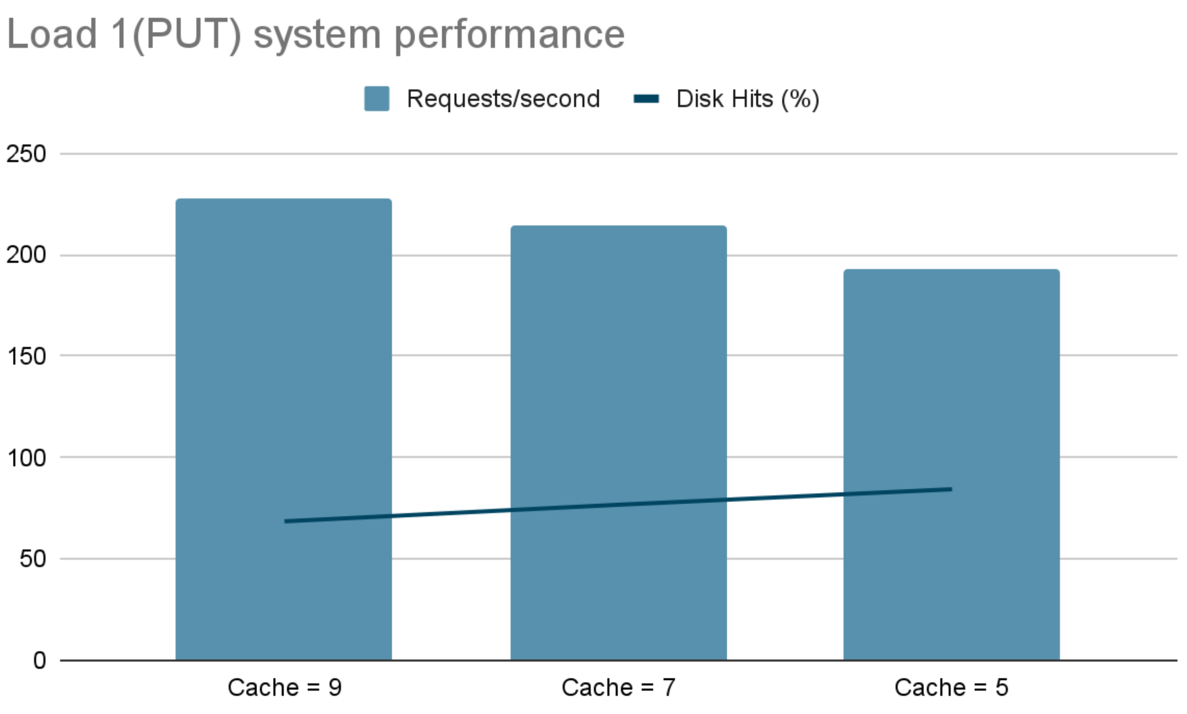
Total weight: 27 points

- **Well formatted document:**
  - We have formatted our document in a way that is easy to read, and we have included diagrams to make it easier to understand. We have included all the sections that the marking scheme requires for the report and labeled them accordingly.
  - The **diagram** here is the entire document :)

Discussing the system's performance (4 pts)

- **Load1:**

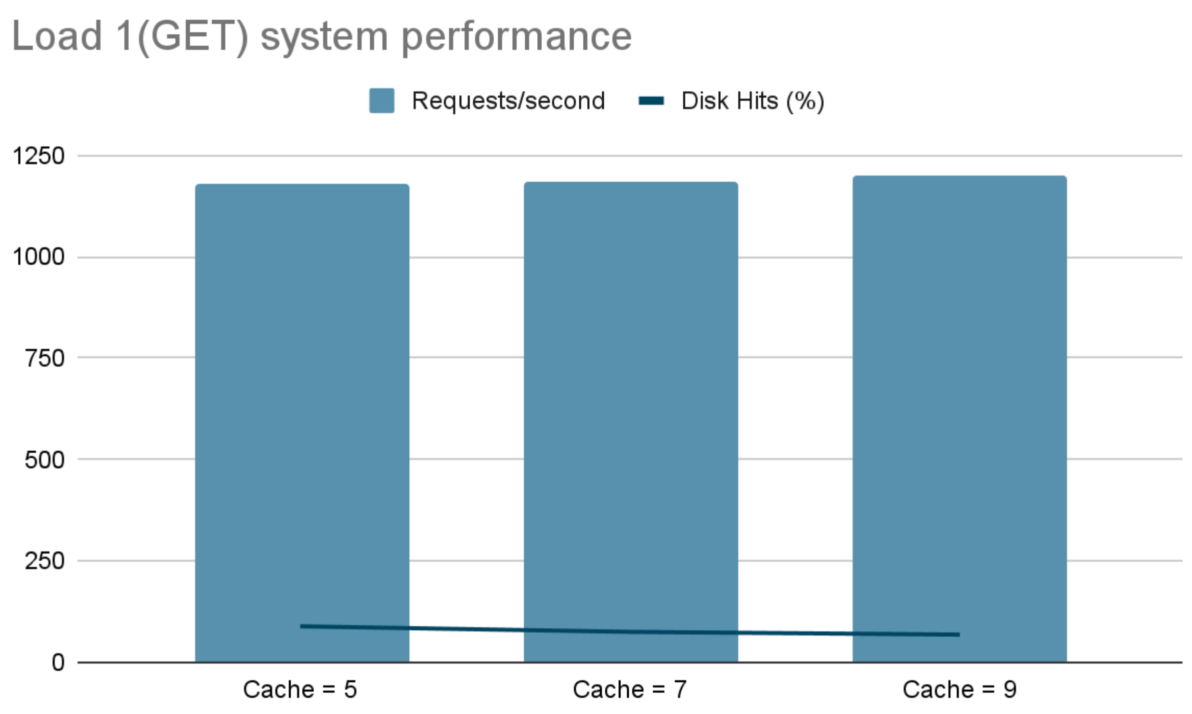
Our first load is a repeat of 50 different PUT requests(100 times) i.e.5000 requests in total. Below is a graph showing that we tried multiple cache sizes to see how the throughput varies. As you can see, when the cache size increases, we get less disk hits, and our throughput increases. This is good, and shows that our code actually works :)



Total weight: 27 points

- **Load2:**

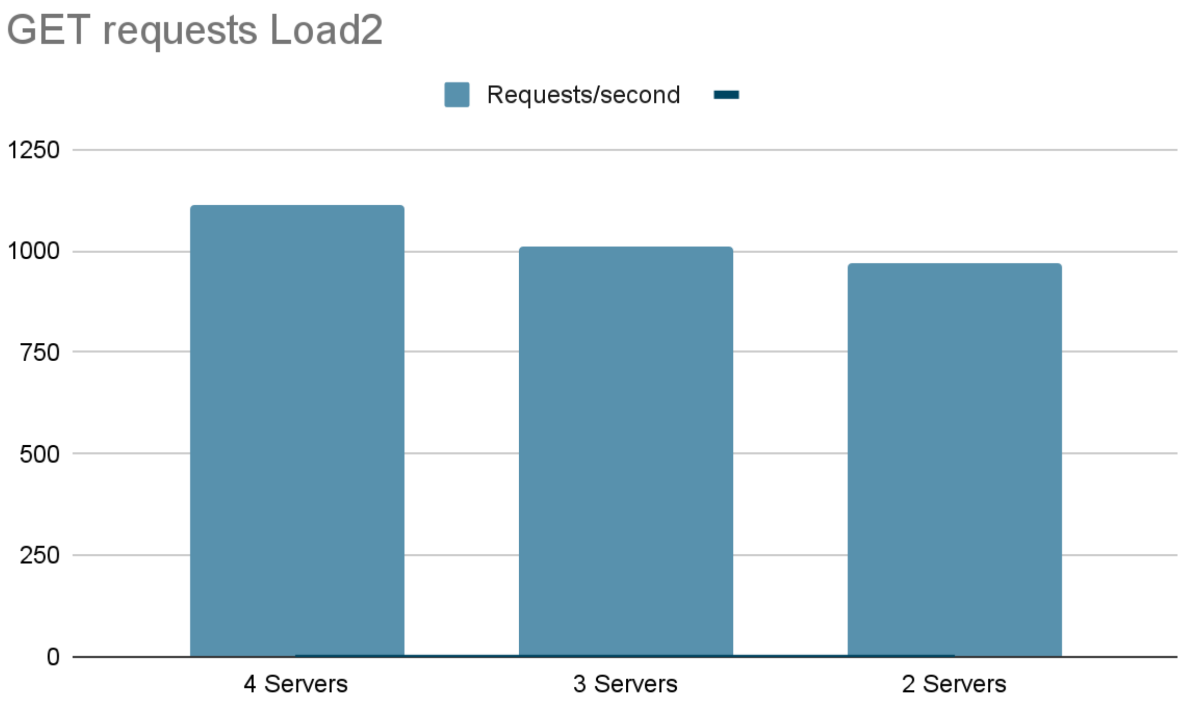
Our first load is a repeat of 50 different GET requests(100 times) i.e.5000 requests in total. Below is a graph showing that we tried multiple cache sizes to see how the throughput varies. As you can see, we get a lot of throughput for GET requests because of the way we communicate with our database. There isn't much variance, even though there is a lot of change with the cache hit percentage(89%, 75.36%, 68.58%)



<u>Total weight: 27 points</u>

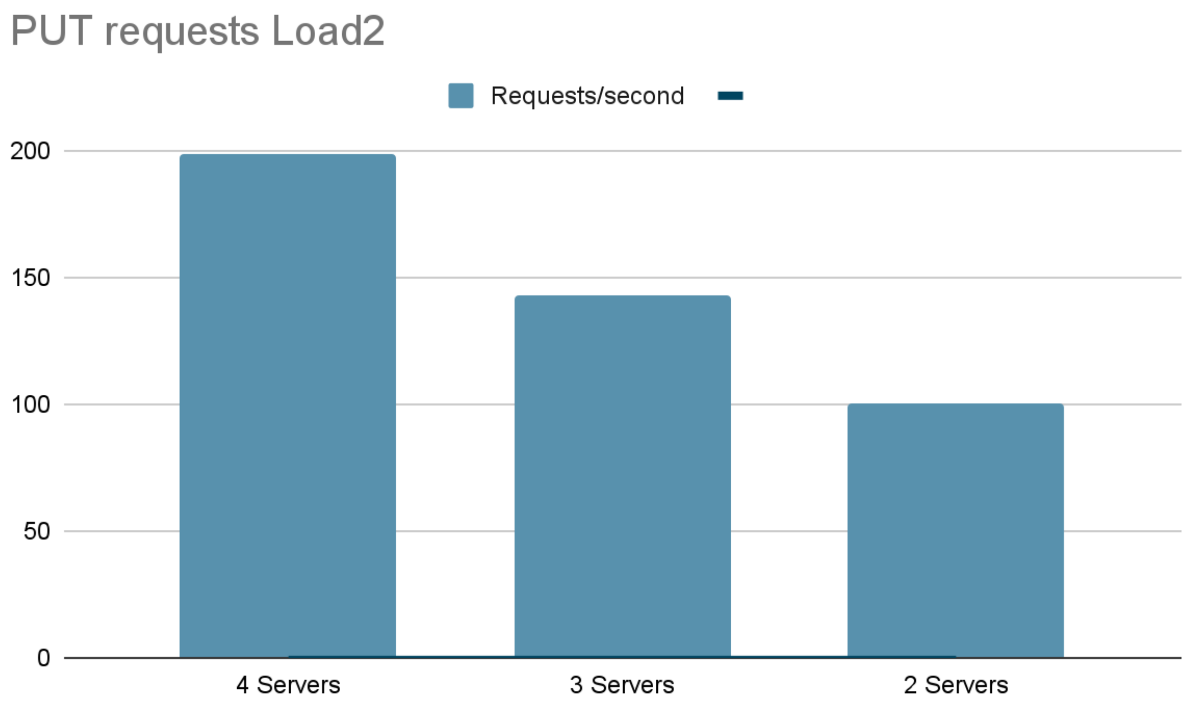
Discussing the system's scalability (4 pts)

- Number of hosts vs read requests per second
    - In the graph below, we see that there is a not so significant decrease in performance for GET requests when we decrease the number of servers. This is to be expected though, because our GET throughput is extremely high.



Total weight: 27 points

- Number of hosts vs write requests per second
    - In the graph below we can see that there is a significant performance drop when we decrease the number of servers. Because PUT is an expensive operation, this is to be expected, although it is unfortunate.



- Number of hosts vs data stored

We are not quite sure what this even means, but we will assume that it means that we run a different number of hosts with different amounts of data pre-stored in the database. We tested this, it really does not change much at all.

Total weight: 27 points

Discussion of the tools used during testing. (2 pts)

Listing them is not enough you must define each tool used, and how you used it

- **Tool1:** JMeter
  - **Definition:** “The Apache JMeter™ application is open source software, a 100% pure Java application designed to load test functional behavior and measure performance.”

  - **Use:** We used this application to test our performance, and to load the graphs when discussing the system’s performance

- I don't think anything else we used would be considered a tool