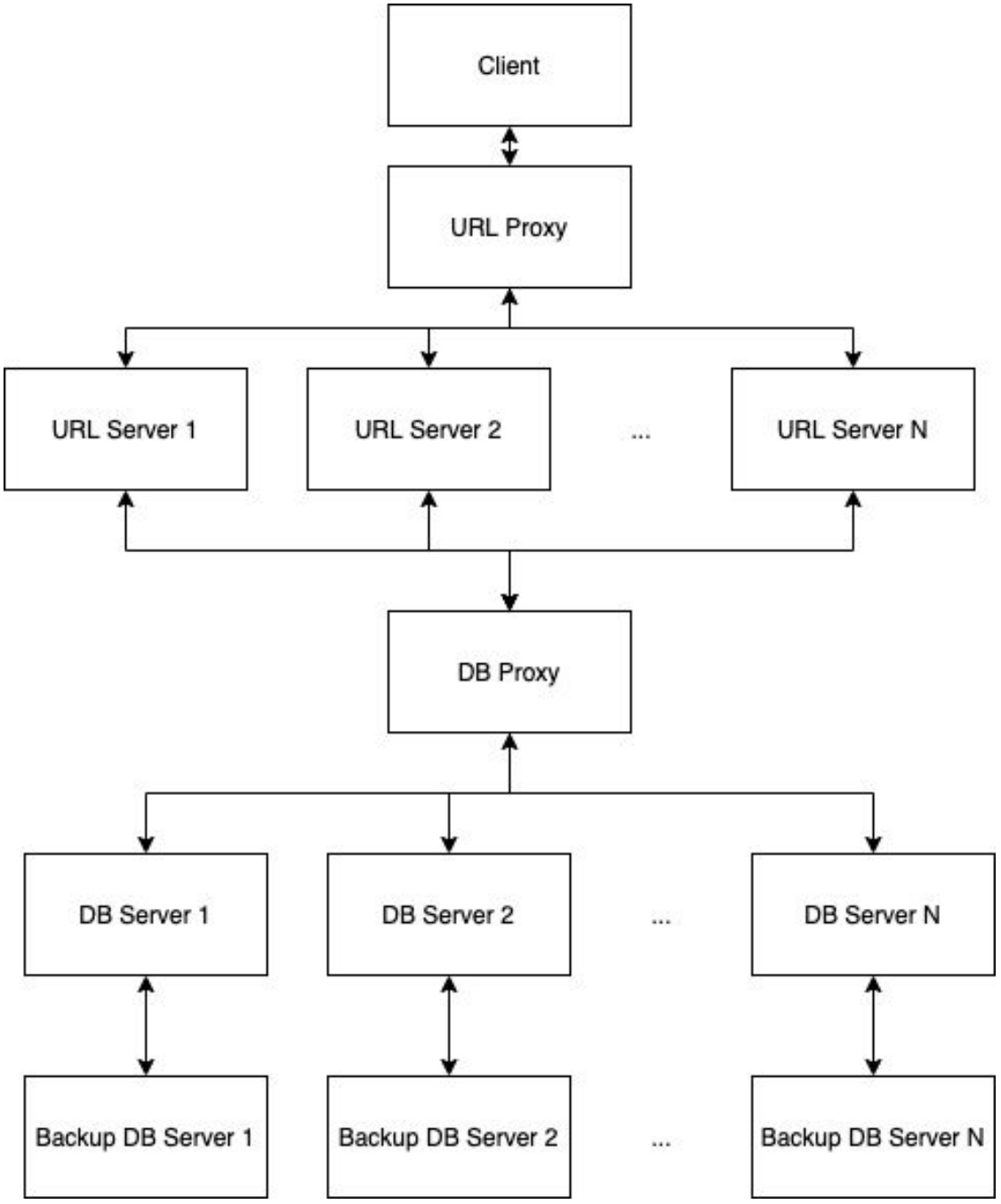
CSC409 Assignment 1

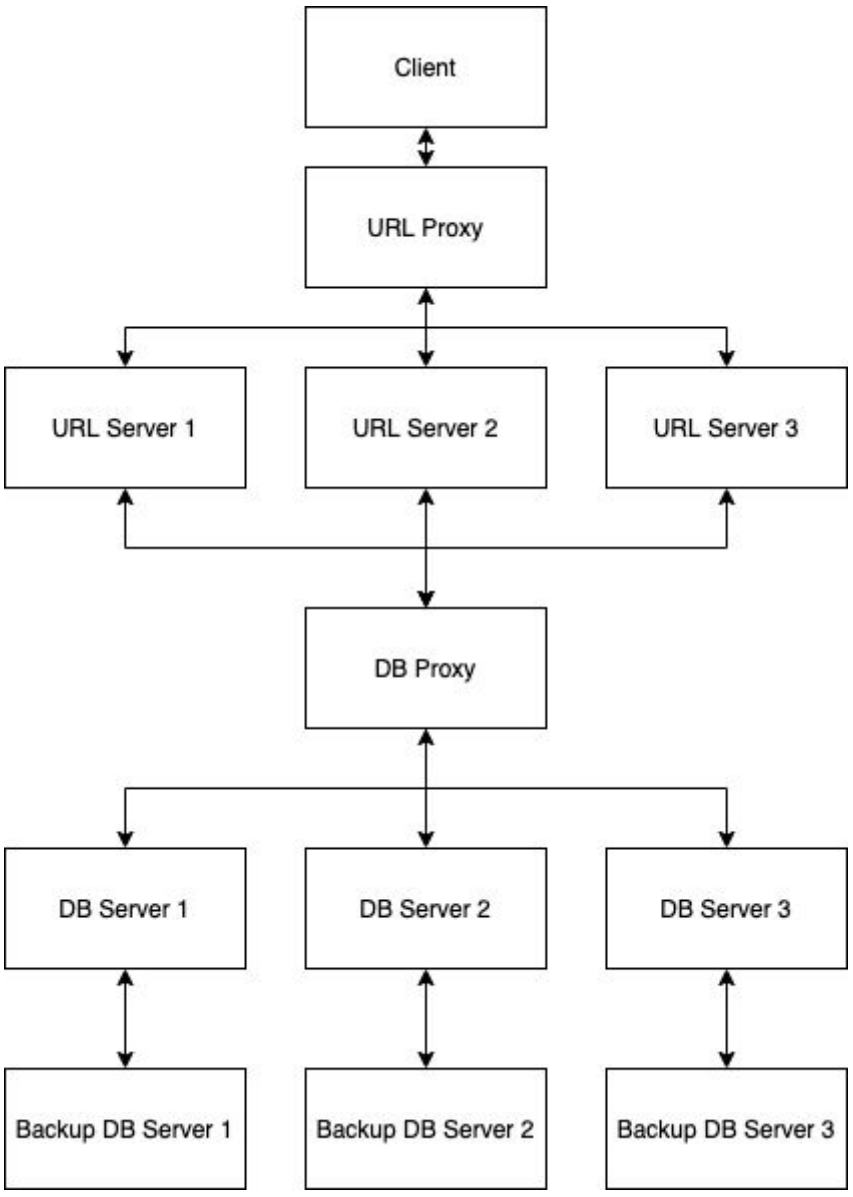
Aashdeep Brar, Abdulwasay Mehar and Vaishvik Maisuria
utorid: braraash, meharabd and maisuri9

Oct 11, 2020

Architecture and Analysis



As seen above, we can have any number of URL Servers and DB Servers running. The diagram below is an example of our service running with 3 URL Servers and 3 DB Servers.



In order for the user to communicate with our system, they will be sending requests to the URL Proxy. The URL Proxy is a multithreaded service that is responsible for load balancing the requests received by any number of clients. The load balancing is done in a round robin fashion where it forwards the request received by the client to one URL Server.

If the DB Proxy tries to connect to a URL Server but is unable to do so, we don't discard the request. What we do is keep trying the next ordered URL Servers until we reach one that works. In the meantime, we have a background monitoring script that is running to check if every single service is functioning. If a service is down, such as a URL Server, it will bring it back up as soon as possible. The monitoring script knows where every single service is running as this is defined in a centralized config file called config.properties.

The URL Proxy is also reading the config file once a minute to detect any changes within the property in the config file that URL Servers are defined. If a new URL Server is added to config, the URL Proxy will pick up on this change and automatically start sending requests to the new server. Likewise, if a URL Server is removed from this property, the URL Proxy will stop sending it requests. This is an example of horizontal scaling as we can add any number of URL Server nodes in an easy manner.

The URL Server contains the starter code that was given to us but is modified to also be multithreaded. Also, instead of communicating with the database.txt file that was provided, it now communicates to the DB Proxy. The database of choice was SQLite3 where it stores the short and long values.

Another large change that was made to the URL Shortener starter code is the addition of a cache. The cache stores any successful response of a GET request for a minute. The advantage of including a cache is that the request that a client makes does not have to go to the database. The URL Shortener retrieves from the cache which stores the values in memory, sends the response back to the URL Proxy which then forwards it to the client. Usually the disadvantage with a cache is that if a PUT request was sent to change the value the short maps to, then the long value in the cache would not be valid. The user would always see this value until the cache stops storing it. However, we believe it is a security risk to override short values as any user can override another user's short, long pair. As the values cannot be overridden, this disadvantage of the cache is negated.

Once the URL Server receives the request from the URL Proxy, it will send the request off to the DB Proxy. Similar to the URL Proxy, the DB Proxy is a multithreaded service that is responsible for load balancing the requests received by any number of clients. The method in which it load balances is by partitioning the data between the DB servers that are running. The way that the data is partitioned is by adding the ordinate values of each character in the short string and modding it by the number of DB servers that are available. For example, if the short value was "hi" and we had 3 DB servers available, we would do the following operation

sum = ord("h") + ord("i") = 104 + 105 = 209
location = sum % DBServers.length = 209 % 3 = 2

This means it will go to the DB on index 2. If we were to send a GET request, the request will be first sent to the main DB and if that is down, it will send the request to the backup DB. The backup DB is running on a separate computer and has its own DB file. The advantage of this is that if the PC that is running the main DB was to get shut down or gets corrupted, the backup DB would not get affected. If the client sends a PUT request, this request is sent to both the main and backup DBs in order

for them to be synchronized. Hence, the system is able to replicate the data instantly.
The main DB and its backup could become unsynchronized if one of them was to ever go down. Due to this, our monitoring system will check if either the main or backup DB is down. If the main DB is down, every single PUT request will be sent to the backup, meaning that it will have up to date information. Before the monitoring script attempts to bring the main DB server back online, it will copy the DB file in the backup to the main DBs computer. Thus, when it's brought back up, both the main DB and backup DB will be synchronized. The process is the same for when the backup DB is down and the main DB is running.

Due to our partitioning method, every single DB has distinct values from other DBs. The advantage of this approach is that GET requests will take a lot less time as they are searching a portion of the data instead of all the records. Also, if a DB and its backup were to get corrupted, we will only lose that partition's information. If we had one large backup that stored all records, we would have a single point of failure. However, this approach introduces the complexity of repartitioning the data if any DB Server was to be removed or added.

Similarly to the URL Proxy, the DB Proxy is reading the config file once a minute to detect any changes within the property in the config file that DB Servers and backup DB Servers are defined. If any server or backup server was to change, we would need to repartition the data as the mapping scheme that is defined (adding all characters in a short and modding it by the length of servers available) will most likely point the request to a server that does not have the correct information.

Whether it be adding or removing a DB Server, once the DB Proxy recognizes a change has occurred, it will send a repartition request to all of the DBs. If a DB is down, it will send the repartition request to the backup DB. Once the DB Server picks up the repartition request, it will retrieve all rows from its table, purge out the table and send its rows to the URL Proxy. While the DB system is repartitioning, we temporarily stop all client requests to get up to date information. The advantage of doing this is we are ensuring that all hosts get the correct values. However, the obvious disadvantage is the system won't be functioning for clients during this small period. While the DB Proxy is receiving all of the rows in the DB Servers, it will start sending each row to the new location. In order to speed up the process, we are sending all requests from the DB Server to the DB Proxy in batches.

When designing the architecture, we gave more priority over to preserving data, instead of speed. Since url-forwarding is a service that prioritizes the handling of user get requests, our architecture works towards padding data loss via system failure, while providing an efficient method for the users to get the required short-long pair.

Running The System

In order to simplify the process of running the system, we have set up a config file which stores where all the services are running. This includes both the host and port. For example, if there was a backup DB Server running on DH2020PC25 port 7000, the db.backupHostsAndPorts field in the config will have dh2020pc25:7000 within it.

The options are either running each service manually or using the startup script. If we're running each one manually, we need to run each service on the same hosts and ports specified in the config file. This is because our services use that file and assume each service is running on the correct host and port.

The other option is using the startup script which will ssh into the hosts specified into the config file and run them in the background. The startup script also has other options such as starting certain services, stopping all services, stopping certain services and deleting all class files.

Testing The System

In order to stress test the system, we used scripts to allow for different configurations. There can be multiple users sending requests at the same time in a real world setting. A script allows us to send requests in separate threads, which mimics what would happen if users were interacting with our system.

The tests were conducted with the following configurations:

1. 2 DB Servers and 2 URL Servers
2. 3 DB Servers and 3 URL Servers

For each configuration, we tested the following with each of them running 10 asynchronous threads hitting the server at the same time:

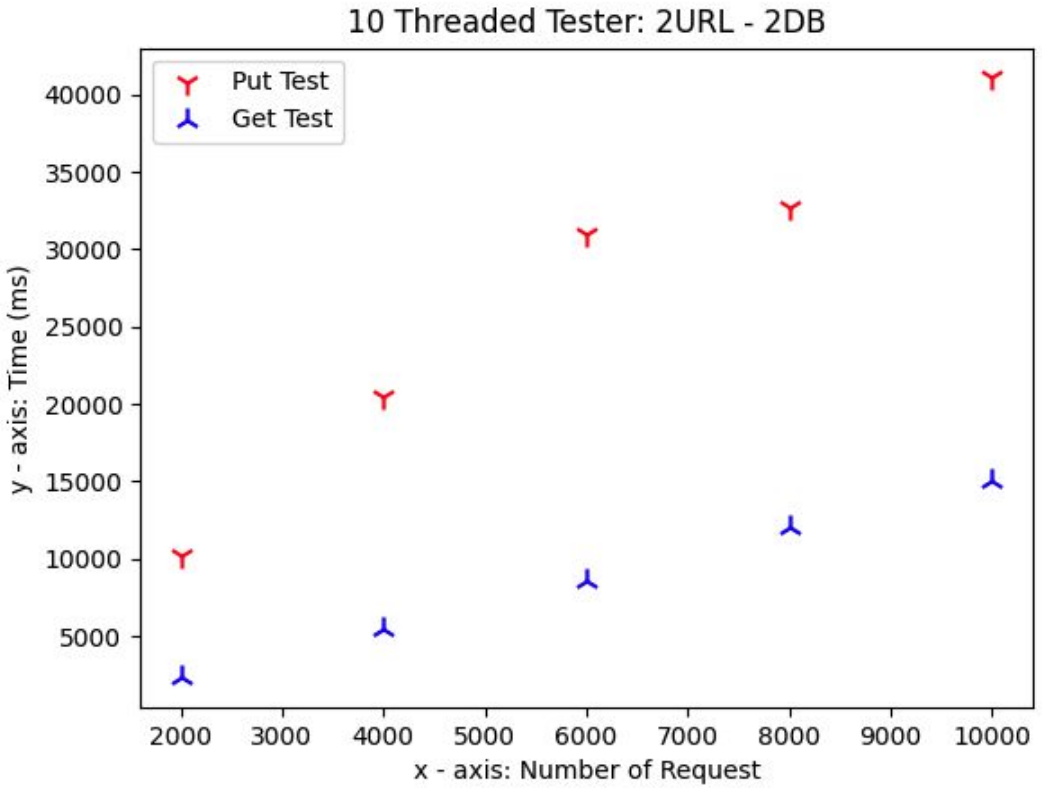
- 1000 requests
- 2000 requests
- 4000 requests
- 6000 requests
- 8000 requests
- 10000 requests

As there are both PUT and GET requests a client could make, we tested each number with both types of requests

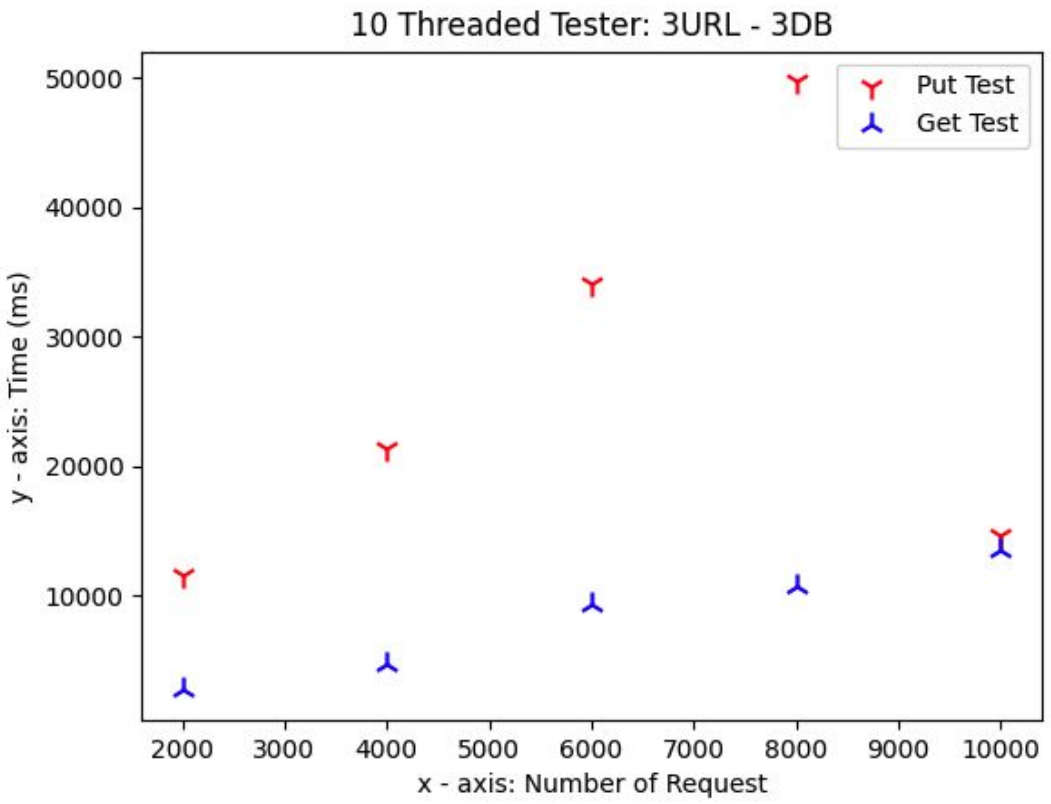
We will measure how the system will function in terms of time, as the number of requests increase. Doing these tests will allow us to see how the overall throughput will change, as each of the clients start to send more requests.

Note, the cache was not used when measuring throughput.

2 DB Servers and 2 URL Servers



3 DB Servers and 3 URL Servers



Raw Data

PUT Requests

	2URL - 2DB	**3URL - 3DB**
2000 Requests	11196.662999999999 ms	336.275 ms
4000 Requests	21518.487 ms	326.272 ms
6000 Requests	31837.046000000002 ms	361.681 ms
8000 Requests	36741.129 ms	316.108 ms
10000 Requests	47089.793999999994 ms	348.018 ms

GET Requests

	2URL - 2DB	**3URL - 3DB**
2000 Requests	1538.019 ms	132.157 ms

4000 Requests	2911.677 ms	129.136 ms
6000 Requests	9213.347 ms	117.405 ms
8000 Requests	13098.934 ms	173.424 ms
10000 Requests	16009.250000000002 ms	135.02700000000002 ms