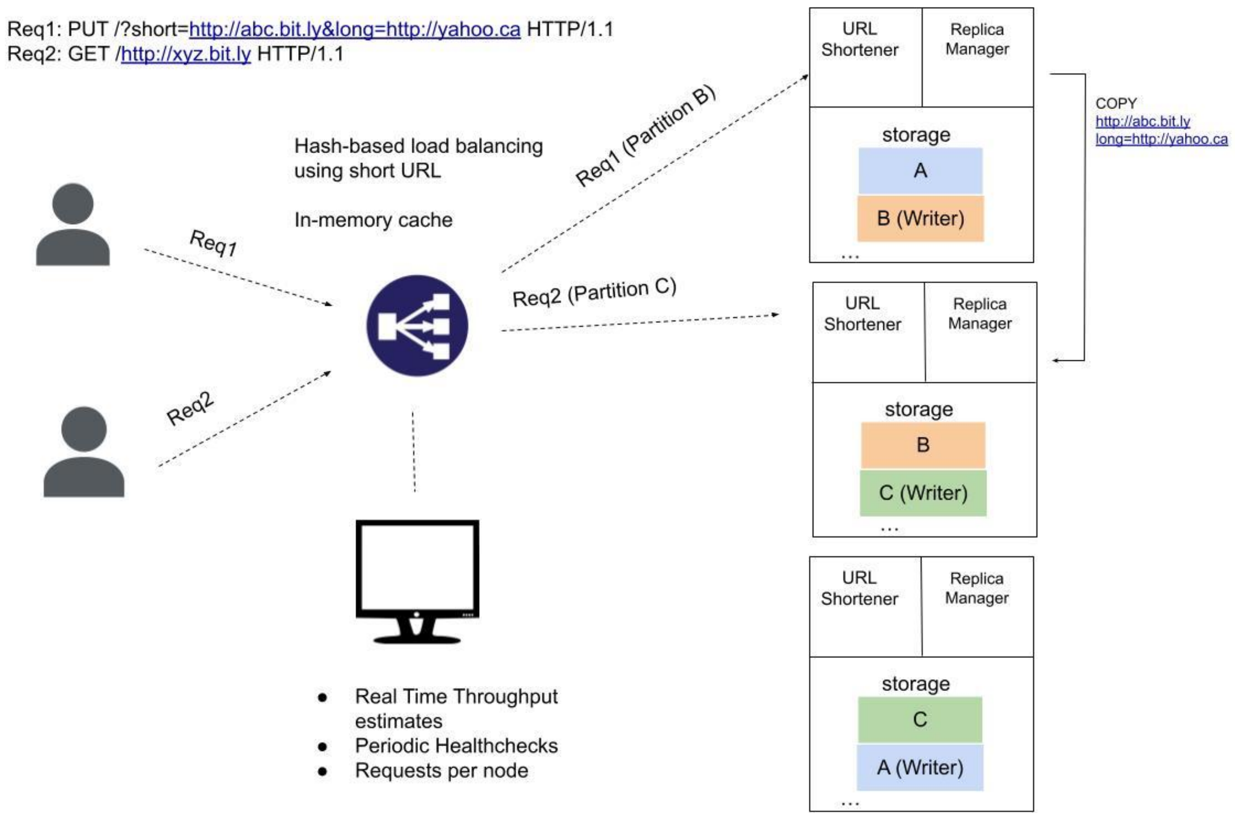
Distributed URL Shortener

CSC409H5, Group 9

Karampreet Daid, Louis Scheffer & Keshav Worathur

Discussion of the System's Architecture

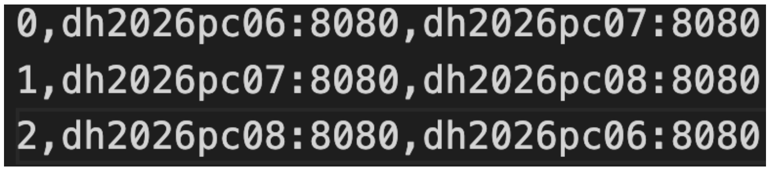
The URL Shortening application allows users to submit pairs consisting of a short and long URL or request the long URL mapped to a given short URL. All requests are made to the load balancer, which provides a point of entry to the service. When a request is received, the load balancer hashes the short URL present in the request to obtain a partition ID. Depending on the partition ID, the load balancer can forward the request to one of the cluster's nodes hosting the URL Shortener service.


Req1: PUT /?short=http://abc.bit.ly&long=http://yahoo.ca HTTP/1.1
Req2: GET /http://xyz.bit.ly HTTP/1.1
Hash-based load balancing using short URL
In-memory cache
Req1
Req2
Req1 (Partition B)
Req2 (Partition C)
Real Time Throughput estimates
Periodic Healthchecks
Requests per node
URL Shortener
Replica Manager
storage
A
B (Writer)
B
C (Writer)
C
A (Writer)
COPY http://abc.bit.ly long=http://yahoo.ca

Consistency (Louis)

The consistency of the service is ensured by using a replica manager which is called directly after the write to the node’s DB succeeds.

The Replica Manager has two major functionalities. It performs this functionality with a pool of threads that is a variable size. The first is that it has function replicate() which is responsible for sending writes to the second node containing the partition. It also has a function called distribute which will read the manifest and determine the number of partitions. The replica manager will then iterate through the database and rehash each shorturl based on the new number of partitions. It will then reference an in-memory version of the manifest and determine which two nodes it should be sent to. In the case that neither node is the current node, then it will then delete the row from the current database.



```
0,dh2026pc06:8080,dh2026pc07:8080
1,dh2026pc07:8080,dh2026pc08:8080
2,dh2026pc08:8080,dh2026pc06:8080
```

Above: an example of the manifest file

In Replica Manager, if the write to the second node fails, we wait 100 milliseconds and try again. If this fails, we note the failure on the node itself. If a node meets our failure threshold in a brief time, we take down the node as its writes are unexpectedly failing. This might mean that there are a few items in the database that are not properly replicated. If we were to automatically make a DISTRIBUTE call without changing the manifest every so often, we could copy any of these un-replicated items to mitigate this issue. Further failures at this point would trigger a server failure and we could restart the system on that node. We set the takedown threshold to 5 failed writes in a row as this is an amount that could not be easily compensated by the DISTRIBUTE call.

At worst 50% of reads of data written within the last minute will fail to return correct data if the copy is not correctly executed. If the system owner decides to expend resources, they can use our system's DISTRIBUTE feature to reduce the chance of this after 1 minute to an extremely small number. We offer this relatively weak guarantee due to the CAP theorem as we have primarily chosen to focus on Availability.

Availability (Louis)

The availability and partition tolerance of the system is also performed by the replica manager and load balancer, each node contains a failover partition for a different database. This means that if a get fails from the primary node of the partition, we send a request to the replica of the partition. This also means that if a node must be taken offline, we can send over the data from that node to a new host.

We use this backup to dynamically change the number of nodes in combination with a distribution process. When a node reads a distribution request, it checks the manifest and determines the new number of nodes. It then iterates through its database and hashes each URL to determine what partition it should now be in. It sends the request to both of these new partition nodes. This system ensures that

For instance, if a heartbeat is no longer detected by the load balancer from a node, we automatically redistribute the data from all nodes to a smaller number of nodes. We also do this upon adding a new node, which is important to allow nodes to not be overloaded and ensure availability for every partition. There may be a brief time where availability is impacted while we dynamically reallocate database rows due to the substantial number of copy requests happening as each of the DBs migrates to where it needs to be. This tradeoff is necessary to ensure the availability and nimbleness of our systems in the long term.

The load balancer presents a single point of failure for the system as in the case of a hardware or process failure on the load balancer's host, users will not be able to access the service.

Partition Tolerance (Louis)

We guarantee that at least one copy of the data is always available except in the case of a failed copy, followed by a network failure. This is minimized should a service owner choose to make distributed calls at a regular interval. In this case, they could virtually eliminate that possibility within 1 minute of a successful put.

In the worst case if 2 nodes go down, we do not guarantee any data retention. We could have a substantial data loss if both the nodes for a singular partition are lost, and their database is unrecoverable.

In the case of a network failure, we guarantee and maximum of 5 inconsistent puts. The failure threshold would identify an issue with the network and reboot the node on our system. This may lead to a temporary availability loss while the network is down. In the case of a complete network failure of our hosts, we do not guarantee any put or get requests.

Data partitioning (Karam)

Each partition stores a different collection of key-value pairs, where each key is a short URL and each value is a long URL belonging to the short URL. The number of data partitions equals the number of nodes hosting the URL Shortening service. The partition in which a short and long URL pair is stored is determined through the DJB2 hashing algorithm [1] which was chosen in part due to it's speed and ability to approximate a uniform distribution of pairs across partitions.

We have multiple databases to store all our data. Every host has two separate partitions that have 100% of the data of those partitions. For example, if we have Host 1, Host2 , and Host 3, then we have DB1, DB2, and DB3. Host 1 has the entirety of DB1 and DB2, Host 2 has the entirety of DB2 and DB3, and Host 3 has the entirety of DB3 and DB1. Each host hosts two full partitions of data instead of having some partial partitions of data. We believe that this is the most optimal design as we already have replication to preserve data in an emergency case. In addition, our extremely fast short-url hashing algorithm can quickly find which host has the data we are looking for.

Data Replication (Louis)

Partitions are duplicated across nodes, and for each of the partitions there is one node assigned to perform updates to the partition. The data replication is managed by a replica manager replica manager process running on each node. Every time a write occurs to the primary node for a partition, we then send a copy of the write to the second node for that partition. This leaves us with a replication factor of 2. We chose this number to avoid overwhelming our system with replication calls between nodes, since we need to ensure consistency between all copies of a partition and then we would need to revert calls more than once. This also led to the possibility that if a removal failed that we could end up with inconsistent state between nodes.

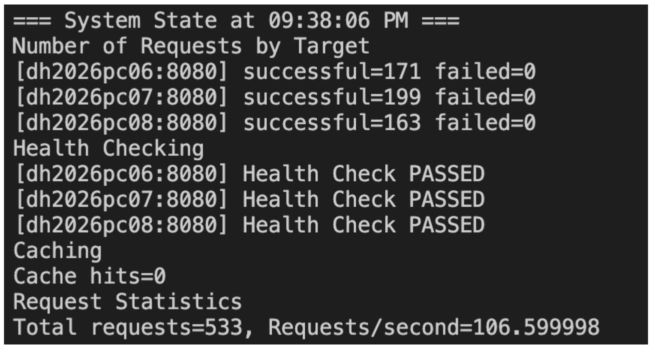
We do not guarantee that data is replicated until a server owner calls the DISTRIBUTE feature, as there could be a failure to process the replicate request. We do guarantee that we will make at least 2 attempts to replicate it before this time and will replace the node if 5 replications in a row fail to process. If the service owner calls DISTRIBUTE every minute, we replicate that all data older than one minute will be replicated.

Load balancing (Keshav):

The responsibilities of the load balancer are broken down into request handling and application monitoring events.

- Request Handling: The procedure for processing and forwarding a client request is replicated using threads to ensure that multiple HTTP requests can be serviced concurrently, achieving the desired request throughput. To distribute the load evenly, hash-based load balancing is used with the DJB2 hash function. This hash function was chosen for its ability to obtain a r

- Application Monitoring: The application monitoring interface provides up-to-date information about the health status of each of the nodes, real time throughput estimates, and the distribution of requests across targets. A sample of the monitoring UI is shown below:



```
=== System State at 09:38:06 PM ===
Number of Requests by Target
[dh2026pc06:8080] successful=171 failed=0
[dh2026pc07:8080] successful=199 failed=0
[dh2026pc08:8080] successful=163 failed=0
Health Checking
[dh2026pc06:8080] Health Check PASSED
[dh2026pc07:8080] Health Check PASSED
[dh2026pc08:8080] Health Check PASSED
Caching
Cache hits=0
Request Statistics
Total requests=533, Requests/second=106.599998
```

Caching (Keshav)

A cache which stores all of its entries in RAM is used on the load balancer to cache responses for recent requests. A short URL is cached with its response after the first request for the URL and stays in the cache for a fixed time interval before the cached response is deleted. We chose this caching policy to ensure as the cache data should be able to fit completely in the machine's physical cache levels and main memory.

Process disaster recovery (Karam)

Process disaster recovery starts with a constant health check. When the health check fails on any URL Shortener service on any of the hosts, we add a new host and spin up a new URL Shortner service. Once the new URL Shortener service is running we send a special distribute request to all URL Shorteners so they can send their data to the newly created URL Shortener service.

Data disaster recovery (Karam)

Our design also supports data disaster recovery if any database was to get corrupted or go down for any reason. Every URLShortner keeps track of how many database failures there has been. If at any point there has been N database failures in the last 2N requests, URLShortner will exit. As a result of this exit, the health check will fail and a new instance of URLShortner will be created like mentioned above. After it's created the other URLShortner's will redistribute their data to each other.

After both scenarios there will be two different replications of data on each machine.

Orchestration

The system achieves administrative scalability through scripts to achieve the following tasks with minimal intervention from an administrator:

- Start the cluster with an initial list of nodes
- Add a new node to scale the cluster out
- Remove a node to scale the cluster in
- Add a node on standby to replace a failed node in the future

These bash scripts will communicate with a separate admin server running on a separate port. Any changes to the state of the cluster are recorded in configuration files to give users an up to date view of the nodes being used by the system.

Health Checking

Every 5 seconds the system is running, the monitoring process will send a GET request for a short URL to each of the active nodes. For each request, the monitor waits up to 5 seconds to receive a non-empty response before declaring the node as inactive. Therefore, requests which are to be forwarded to an inactive node within 5 seconds of this node's process or storage failure will fail and return an empty response. After this point, a standby node will take the place of the failed node.

Horizontal scalability

Horizontal scalability is achieved through scripts to quickly increase/decrease the number of nodes running the URL Shortening service. As the system can support an arbitrarily large number of nodes, this allows compute and storage capacity to be scaled without limits.

Vertical scalability

Vertical scalability can help our system run much faster. We make extensive use of multithreading throughout our design, as a result if we were able to upgrade to a CPU that had more cores our application would be able to process more requests simultaneously and as a result have more throughput.

We also have an in-memory cache in our load balancer. If we could increase the amount of memory available on each machine, we could host a larger cache resulting in fewer misses and an overall faster response time for the client.

Our design uses SQLite3 as persistent storage. If we increased the available amount of hard drive space, our database could store more information.

Analysis of the System's Performance

To test the performance of the system, we subjected it to two loads:

Load1: 4000 GET requests from 4 clients where the requests are for two short urls that map to each of the three partitions used in testing.

Load2: 4000 PUT requests from 4 clients where the requests update 4 short URLS that map to each of three partitions used in testing

The latency histogram for load1 is shown below:

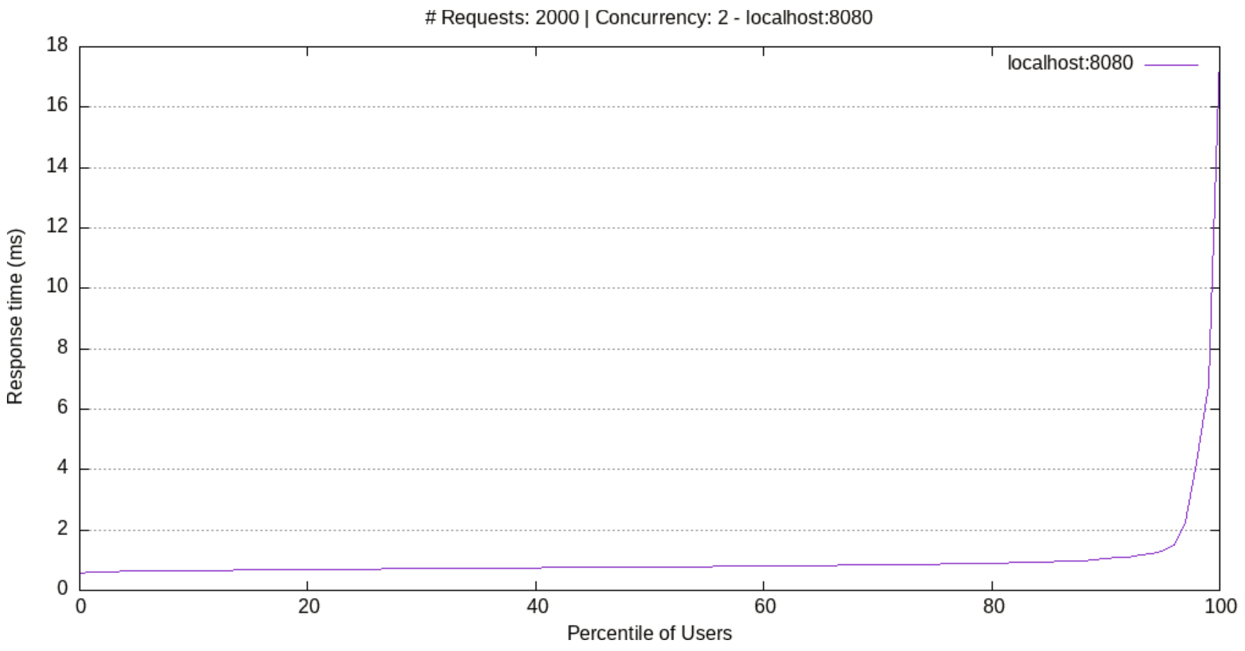


Figure 1: Response Times by Percentile of Users under Load 1

Percentile	50%	66%	75%	80%	90%	95%	98%	99%	100%
MS delay	1	1	1	1	1	1	4	7	18

The system can serve 95% of GET requests within a 1ms response time. This test also only uses two of the three available hosts, this means there is still additional bandwidth left for additional requests.

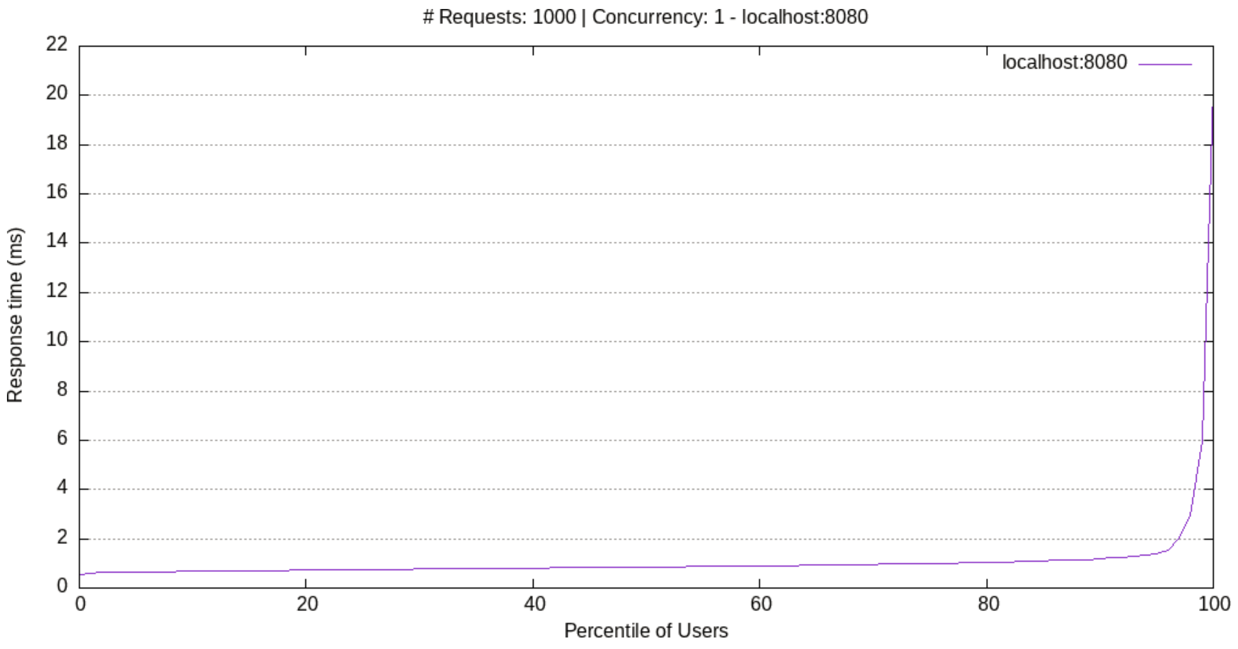


Figure2: Response times by Percentile of Users for load2

Percentile	50%	66%	75%	80%	90%	95%	98%	99%	100%
MS delay	1	1	1	1	1	1	3	6	20

The system can process 95% of PUT requests within 1ms, as we get closer to the 100th percentile performance degrades slightly.

In both cases we can make a guarantee that up until around the 95th percentile there will only be a latency degradation of a fraction of a millisecond. As we get close to the 100th percentile the level of degradation increases significantly to around 18-20x of the 95th percentile case. We cannot make a low millisecond guarantee for the last 5% of cases. We believe the overall performance of this system is more than adequate. Being able to serve 95% of requests without more than a fraction of a millisecond in latency is very good.

Analysis of the System’s Scalability

The system was designed to be highly scalable when given additional nodes to run on. The system is also scalable when given more resources on each of the nodes, but less so than when scaling the number of nodes.

When given one or more new nodes, we can execute a script which will automatically start the URL Shortener service on the node, update the manifest file, and redistribute the existing pairs of urls.

We have several parameters that can be changed when necessary to make use of the resources of the system located within the URL Shortener and the load balancer. Examples of these on the load balancer include thread pool size and cache duration. If the load balancer node receives more CPU cores or threads, we can change the thread pool size to make use of this. Likewise, we can do this with the URL Shortener if it receives more CPU cores or threads.

When we scale up the number of hosts we see an increase in throughput of roughly 250 writes per second and 750 reads per second until we max out the speed of the load balancer. We can also scale up the storage capacity by the number of nodes. For practical purposes, each node cannot have more storage than ¾ of the nodes memory capacity as the high scalability of the system relies on heavy memory use to server requests quickly and adapt to changes in the system. In theory it should be possible to store more on each node, but it would severely impact the performance of the system, so we do not recommend it. If we cannot store data structures in memory, we would not be able to quickly redistribute the pairs when nodes come online or go down.

Testing the System

The hosts used for testing have Intel core i7-9700 processors, with each CPU having 8 cores.

Apache Bench[2] is an HTTP server benchmarking tool. It allows a load tester to send thousands of requests to a HTTP web server with requests being sent concurrently, simulating multiple users. The tool collects useful information such as the percentage of requests falling within a certain latency and average request throughput.

GNUPlot [3] is a command-line graphing utility for Linux. To create plots from benchmarking data, we used an open-source helper script [4] for producing the response latency by percentage of users plots from apache bench csv output.

We use bash scripts to test the failure recovery of the system. In the failure recovery test, we start the system, add a standby node, then kill one of the currently active URL shortener nodes to test the system’s ability to respond to a host failure.

References

[1] “Hash Functions” http://www.cse.yorku.ca/~oz/hash.html Accessed 2023-10-11

[2] Apache Bench: https://httpd.apache.org/docs/2.4/programs/ab.html

[3] GNUPlot: http://www.gnuplot.info/

[4] "apachebench-graphs". Courtesy of Juan Luis Baptiste: https://github.com/juanluisbaptiste/apachebench-graphs