Design Space Exploration of URL Shortening Service

Dhruv Deshmukh
dhruvr@iitbhilai.ac.in

Harsh Vardhan
harshv@iitbhilai.ac.in

Abdurrahman Khan
abdurrahman@iitbhilai.ac.in

Shubham Gupta
shubhamgupta@iitbhilai.ac.in

Shashwat Jaiswal
shashwatj@iitbhilai.ac.in

Gagan Raj Gupta
gagan@iitbhilai.ac.in

1. INTRODUCTION

Design space exploration is an important step in designing any large scale computer system. It allows one to analyze the price, performance and consistency of different design dimensions available and aids in choosing the design giving optimal performance per unit price while meeting all application requirements. Our current works endeavours to analyze a few different design choices for a URL Shortening Service and test their performance at scale. The setup and implementation of the choices in consideration has been done using the AWS infrastructure. We have tried to include as diverse and commonly used design choices as possible so as to make this study inclusive of answers to questions that modern system designers may face. We have compared metrics like response time, throughput and price per transaction for these designs.

URL Shortening Service has much in common with larger and more complex systems. Specifically, it comprises all the basic operations such as load balancing, application server pool, caching, database and analytics that any software system may offer to its users. One can build upon these basic services to make complex systems like Dropbox and Social Media Apps. Towards this end, we discuss various insights gained through our experiment and possible improvements to the system and design choices.

One can find several Design Space Exploration research papers for various types of systems but surprisingly very few of them have analyzed an application like a URL-shortening service. Many websites present approaches to design a URL-Shortening service but their claims are not supported by experimental results. It is also not clear if the design choices will work at the scale of a real-world service such as Bitly. Our paper takes a more practical approach by trying out different database choices like SQL, key-value pair based NoSQL and document based NoSQL as well as choices such as decoupling read and write servers and caching.

2. REQUIREMENTS AND CAPACITY ESTIMATES:

These requirements and assumptions for the workload and capacity estimation of our systems are in accordance with similar real world systems like Bitly and TinyURL.

2.1 Requirements

- Given a URL, the service should generate a shorter and unique alias of it.
- When users access a short link, the service should redirect them to the original link.
- Users should optionally be able to pick a custom short link for their URL.
- Links will expire after a standard default timespan. Users should optionally be able to specify the expiration time.
- The system should be highly available.
- URL redirection should happen in real-time with minimal latency.
- The shortened links should not be predictable.

2.2 Workload and Capacity Estimates

The average size of a URL has been assumed to be 100 bytes. We assume that 500 million new short URLs are created every month. We assume the read to write ratio to be 100:1 as redirection to shortened URLs will be much frequent than creation of new shortened URLs. Also, we assume that traffic generated follows the 80-20 rule i.e. 20% of the URLs generate 80% of the traffic. The Table 1 summarizes workload and capacity estimates based on the assumptions made above.

Category	Estimate
New URLs	200 /s
URL re-directions	19 K/s
Incoming data	20 KB/s
Outgoing data	1.8 MB/s
Storage for 5 years	3 TB
Memory for cache	34 GB

Table 1: Workload and Capacity Estimates

3. DESIGN CHOICES

We will primarily focus on three design choices in our analysis:

Choice	Choice 1	Choice 2	Choice 3
Language	Golang	EJS	EJS
App Server	Unified	Decoupled	Unified
Load Balancer	Classic	Application	Application
Consistency	Eventual	Strong	Strong
Database	Amazon DynamoDB	Amazon MySQL RDS	Amazon DocumentDB
Redis Cache	Yes	Yes	No

Table 2: Design Choices

Choice 1: Golang is coupled with Amazon's DynamoDB to reach a state of eventual consistency model. The idea behind the eventual consistency is the human eye-hand co-ordination time and screen response time. The Redis Cache improves performance by caching the most frequent URLs. Classic Load Balancer and Golang are a good combination for design of Microservices because of their simplicity and speed.

Choice 2: MySQL provides ACID Constraints and Strong Consistency. The Nodejs library has built-in support libraries to host a MySQL Server. We have two types of requests write(high service time) and read(low service time). Decoupling works best in such a scenario. The Application Load Balancer can be configured to send write requests to a group of servers and read requests to another group thus giving us a Size Interval Task Assignment type policy for task assignment.

Choice 3: MERN(MongoDB, Express, React, Nodejs) is very popular tech stack. The best part of the combination is the use of JSON format for storing and parsing of data in both server(Nodejs) and database(MongoDB). We divert from this stack a bit by using Amazon DocumentDB instead of MongoDB as both use document based NoSQL storage.

4. QUEUING MODEL

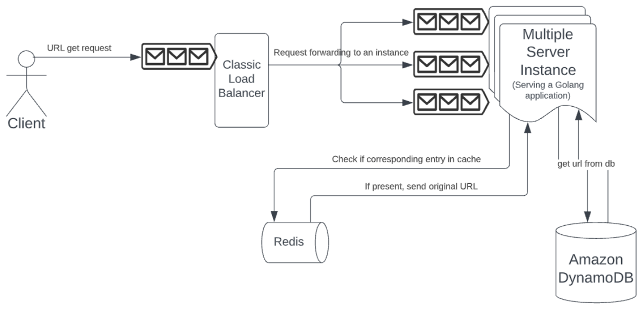
URL get request
Client
Classic Load Balancer
Request forwarding to an instance
Multiple Server Instance
(Serving a Golang application)
Check if corresponding entry in cache
If present, send original URL
get url from db
Redis
Amazon DynamoDB

Figure 1: Queuing cum System Design Model for Read Requests in Choice 1

Figure 1 depicts the anatomy of a read request for Design Choice 1. We assume that write requests are for new URLs and therefore will directly update DynamoDB without needing an update to the cache. In the decoupled model used in Design Choice 2, the application servers will be partitioned into two groups(read servers and write servers). The classic load balancer will be replaced by application load balancer and DynamoDB with RDS MySQL. In Design Choice 3, there is no cache hence while reading the database is visited directly.

The arrival of processes is modeled as a Poisson process. Considering the read-write ratio and the cache-hit and miss scenarios we can categorize job sizes into 3 types given in the table below.

Read/Write	Category	Time Taken	Probability
Write	New	$T_{rdb} + T_{wdb}$	$\frac{1}{101}$
Read	Cached	T_{rc}	$\frac{100 * p}{101}$
Read	Not Cached	$T_{rc} + T_{rdb}$	$\frac{100 * (1-p)}{101}$

Table 3: Category of requests

The above table gives the distribution of incoming job sizes where T_{rdb}, T_{wdb} and T_{rc} are the random variables representing the time to read from db, the time to write to db and the time to read from cache. Let us call this distribution L. So the queue at load balancer can be modeled as M/L/k. The queuing model at each server will depend on the task assignment policy of load balancer. For Classic LB the policy is Join the Shortest Queue and Application LB it is Round-Robin[3].

5. EXPERIMENTAL SETUP

The implementations consist of a Load Balancer configured for receiving HTTPS requests at port 443. A self signed x509 certification is used for TLS implementation. The load balancer distributes the incoming load among a total of 12 t2.micro EC2 instances. In case of design choice 2 the 12 instances are divided into 2 write servers and 10 read servers.

Each of the databases has been pre-populated with approximately 10 million URLs. The URLs are mostly synthetic and have been created using some real URLs as base URLs and appending characters to these URLs. A Redis cache with 1 primary node(cache.t2.micro) and 2 replicas was used in Choice 1 and a Redis cluster with two shards having 2 replicas each was used in Choice 2 . The size of the cache is 0.5 GB. Thus the cache capacity is about 1 million URLs.

For testing, we deployed 5 VMs each sending around 4000 requests/second (RPS), giving about 20,000 RPS as well as 200 writes per second from a single EC2 t2.micro instance as the background load on the system. To mimic an user activity we setup another EC2 t2.micro instance that sends read requests at an average rate of 2 requests. The latency observed by this user is plotted and used for analysis. To have a fair comparison with Bitly in terms of latency this user is setup in the same region in which the server is running. In our case this was the Asia-Pacific South-1 Region and for Bitly it was the US N, Virginia region.

The traffic is modeled as a Pareto Distribution . To generate this traffic we use a tool for load-generation known as k6. The instances generating the background traffic simulate 1000 virtual users such that the net requests/second from that instance is about 4000 as mentioned above while the instance that mimics a user has only one virtual user. After a soak period of 1 hr, the test for each choice lasted for 30 minutes.

6. RESULTS & DISCUSSION

Metric	**Choice 1**	**Choice 2**	**Choice 3**
Latency(avg)	28.17ms	12.36ms	1.23s
Latency(P(95))	112.13ms	37.43ms	3.78s
BAL	251.35ms	361.176ms	797.97ms
Throughput(MB/min)	34.920	22.133	5.049
RPSPI (Avg)	14,162.336	10,585.95	1561.5
PPM[4] (avg)	243.62	268.25	367.16
CPTPS ($\times 10^{-15}$)	1.86	2.04	2.80

Table 4: Table illustrating the performance metrics Note: RPS is Requests per Second, PPM is Pricing Per Month , CPTPS is Cost per Transaction per Second, BAL is average background latency(It has added network delay and queuing delay as many TCP connections on loading servers)

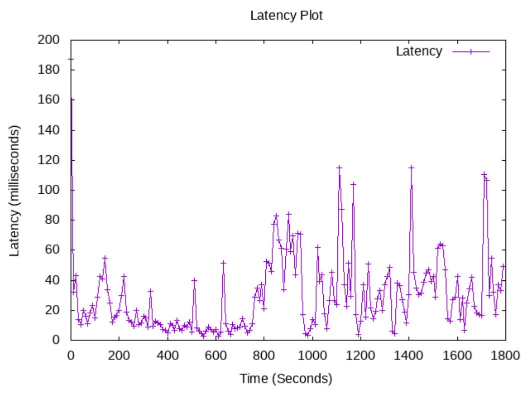

Figure 2: Latency vs Time graph for Choice 1

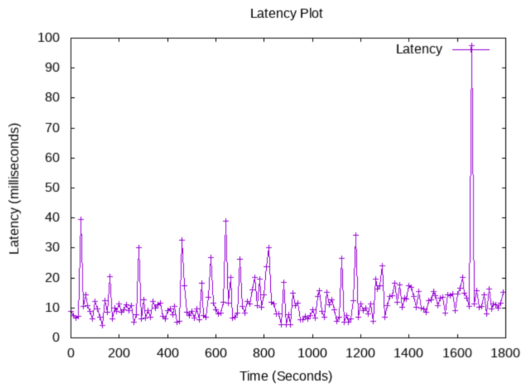

Figure 3: Latency vs Time graph for Choice 2

The results show that the choice 2 performs better in terms of average latency than choice 1. More importantly, choice 1 is much worse than choice 2 when compared on the 95th percentile latency i.e 112ms and 37ms respectively. This can be accredited to the performance of DynamoDB as it provides reliable average low double digit latency but the maximum latency can be in the hundreds of milliseconds or even higher[2]. This gives us a contrast in the latency and throughput capacity of RDS and DynamoDB. Choice 3 didn't seem to perform at all in the present scenario due to no caching. It is the costliest and the slowest performing system of the three. This can be further improved by adding indexes to the shorturl attribute in the db.

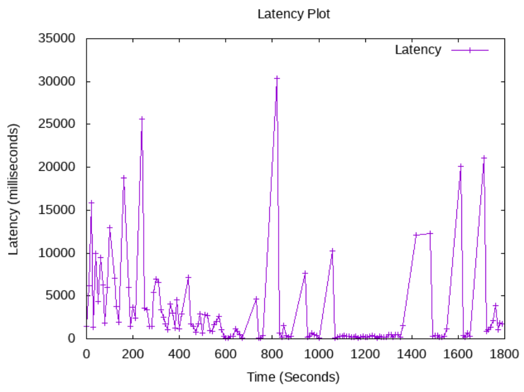

Figure 4: Latency vs Time graph for Choice 3

Choice 1 is the cheapest of the options with an approximate cost of $\$1.86 \times 10^{-15}$ per transaction with Choice 2 as the second cheapest. Design choice 1 however seems to be more sensitive to the network input. It can be observed in Fig. 2 that with slight increase in the load of the server the latency increases.

We also took reading of bitly.com 's shortening and redirection latency from ec2 instances in different aws regions:

Region	US (N. Virginia)	South America (Sau Paulo 2)	Asia Pacific (Mumbai)
Latency(avg)	12.2ms	125.56ms	283.79ms
Ping RTT	0.633ms	1.079ms	1.381ms
Local Time	6:40 AM	7:40 AM	4:12 PM

Table 5: Table illustrating Bitly's perfromance in various AWS regions

It seems that Bitly has deployed its services in all the 3 as the ping RTT is similar for all of them. The load on North-America region seems to be low and the latency is comparable of our deployments. However, the other two regions have latency higher than 100ms ans 200ms respectively.
The deployed and simulated scale loads on our systems comparable to Bitly were achieved. Design choices 1 and 2 seem comparable in the latency and cost metric, however we need to regularize the load on both the design further to obtain a better comparison. The choice 3 needs to be optimised and scaled to handle the load. Further research will focus on improving the design choices based on the observations and regularization of the testing environments.

7. REFERENCES

[1] L. Nardi, D. Koeplinger and K. Olukotun, "Practical Design Space Exploration," 2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), 2019, pp. 347-358, doi: 10.1109/MASCOTS.2019.00045.
[2] https://medium.com/textnowengineering/the-whacking-game-ee3af79c6e13
[3] https://docs.aws.amazon.com/elasticloadbalancing/latest/userguide/how-elastic-load-balancing-works.html
[4] https://calculator.aws//