### **TinyURL System Design**

### How Would You Design a URL Shortening Service?

This is one of the most commonly asked questions for a system design interview!

URL shortening is used for creating short aliases for much longer URLs. For example, goo.gl becomes https://developers.googleblog.com/2018/03/transitioning-google-url-shortener.html. How did that happen! Well, let’s have a look.

Before designing any system, we need to decide on the most basic requirement. So let’s get to it.

### 

### Functional Requirements

* Get a short URL from a long URL.
* Redirect to the long URL when a user tries to access the short URL.

### Non Functional Requirements

Since this service can be used by a lot of companies, it must have both of these NFRs

* Very low latency.
* Very high availability.

There are a few things we need to know before we start building this system.

### What Length Should the URL Be?

This will depend on the scale at which our system will work. For a few hundred URLs, 2-3 characters will be enough, but for something with a larger scale like Facebook or Google, we might need longer URLs. We can either fix the length of the URLs our system will be generating, or we can start with a length and then increment the length as needed. For now, let us discuss the fixed-length approach.

### What Factors Will Affect the Length of Our Short URL?

* Traffic expected - how many URLs we might have to shorten per second,
* The amount of time we need to support these URLs.

Let us look at it mathematically. Say we get X requests in a second, and we need to support these URLs for, let’s say, ten years, then we need to be able to store Y unique URLs at a time where,

Y = X \* 60 \* 60 \* 24 \* 365 \* 10

### What characters can we include in the URL?

Generally, most systems support URLs with a character set consisting of A-Z, a-z, and 0-9. So let's build our URL using this character set. However, in an interview, it might be a good idea to confirm that this is okay before proceeding with your solution.

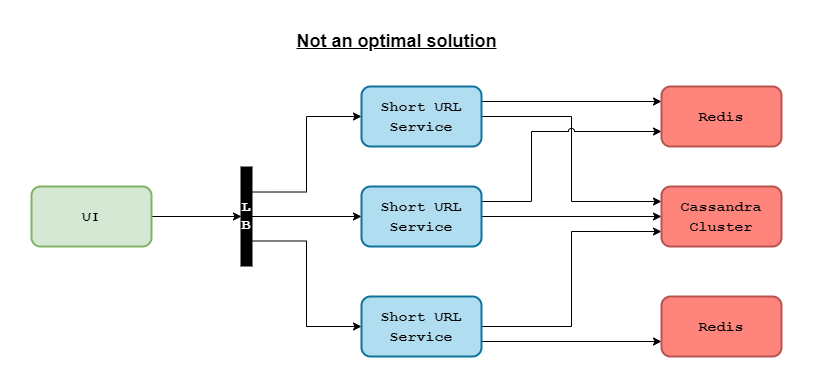
So we have 62 items in our character set which means we need to come up with a length that can support Y URLs with 62 characters.

If the length is 1, we can have 62 URLs.

If the length is 2, we can have 62 ^ 2 URLs.

Similarly, if the length is *L* , we can have 62 ^ L URLs.

Now let us look at a basic architecture that can potentially solve this problem.



### System Architecture

We will have a UI that takes a long URL as an input, calls a short URL service, which will somehow generate a short URL and store it in a database and then also returns the short URL. Similarly, it can also be used to fetch the longer URL when given a short URL. In this case, the short URL service will fetch the long URL from the database and redirect the user to the longer URL.

### More Importantly, How is this Short URL Generated?

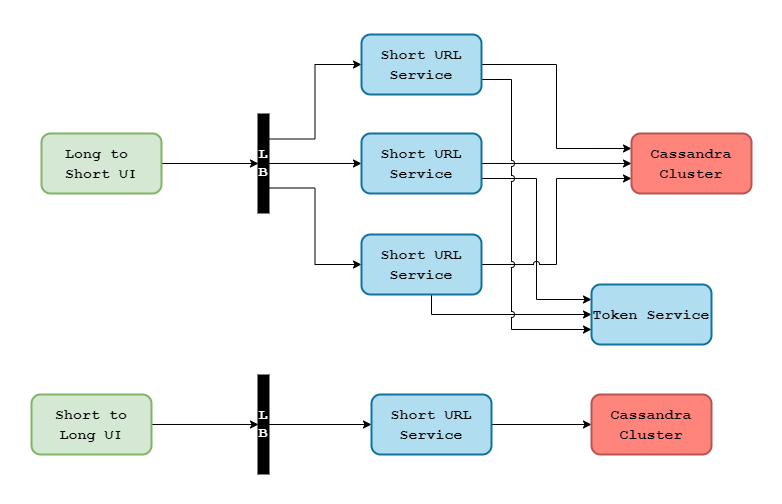
Even though we have 62 characters, for ease of understanding, let's assume that we are generating numbers instead of URLs. So when we say unique URL, for the scope of this article, we mean a unique number. We can safely assume we will be running multiple instances of this short URL service (SUS), say SUS1, SUS2, SUS3, and so on. Now there is a possibility that more than one of them will end up generating the same number, which means one short URL will now point to two long URLs, which we cannot allow. This is known as a collision in computer science.

One way to avoid this would be to check in the database before storing the short URL, to ensure that it doesn’t already exist and retry if it does. But this is a very poor solution. A better solution would be to use a Redis. This Redis will basically start counting from 1 and keep incrementing the count for each request before responding back. With this unique number, we can generate a unique URL by converting it to base 62. This is a very simple way to avoid collisions, but there will be a few challenges with this system.

First of all, every machine will be interacting with Redis, which will increase the load on Redis. Also, the Redis here becomes a single point of failure, which is a big NO! If Redis goes down we will have no way to recover. Another issue is if the load becomes more than what our Redis can handle, it will slow down the whole system.

What if we use multiple instances of Redis? This will give us better performance and higher availability! Until both systems start generating duplicates. We can avoid this by assigning a series to each Redis, but what happens when we want to add another Redis? In this case, we need a managing element to keep track of which Redis has which series. Since we are introducing a managing component, we might as well look into alternatives for Redis.

### Building it for Scale



Our requirement is to ensure our short URL service is generating unique numbers such that even different instances of the service cannot return the same number. That way, each service generates a unique number, converts it to base 62, and returns it. The simplest way to implement it would be to set a range for each service and to make sure each service has a different range. We will use something called Token Service, which will be the managing component in our system. The token service will run on a single-threaded model and cater to only one machine at a time so that each machine has a different range. Our services will only interact with this token service on startup, and when they are about to run out of their range, so token service can be something simple like a MySQL service as it will be dealing with a very minimal load. We will of course make sure that this MySQL service is distributed across geographies to reduce latency and also to make sure it is not a single point of failure.

Let’s look at an example. Say we have 3 short URL services, SUS1, SUS2, and SUS3. On startup say SUS1has the range 1-1000, SUS2 has the range 1001-2000, and SUS3 has the range 2001-3000. When SUS1runs out of its range, token service will make sure it gets a range that hasn’t been assigned to another machine. One way to ensure this would be to maintain these ranges as records and keep an “assigned” flag against them. When a range is assigned to a service, the “assigned” flag can be set to true. There could be other ways to do it, this is just one approach that can be followed.

### How Do We Scale It?

Now we mentioned using a MySQL service that handles a very low load. But what if our system gets bombarded with requests? Well, we can either spin up multiple instances of the MySQL instances and distribute them across the map, as mentioned previously, or we could simply increase the length of our range. That would mean that machines will approach the token service at a much lower frequency.

### Missing Ranges

So we now have a solution that gives us unique numbers, is distributed around the world to reduce latency, and doesn’t have a single point of failure. But what if one of the services that haven’t used up the complete range shuts down? How will we track what part of its range was left unused? We won’t! Tracking these missing ranges will add a level of complexity to our solution. Now, remember that we have more than 3.5 trillion possible unique numbers, which is a huge amount compared to the few thousand unique numbers that we are losing. A few thousand numbers are not significant enough to complicate our system and possibly compromise the performance. So we will just let them go and when the service starts back up we will assign it a new range.

### How Will We Redirect a Short URL to the Longer URL?

When a request arrives for a short URL to be redirected to the long URL, we will fetch the long URL from the database, the service does a redirect, and the user ends up on the right page. Now, if you look back at the diagram, we have used Cassandra as our database. We could have used any database that has the capability to support 3.5 trillion data points, it can also be done with a MySQL database with a little bit of sharding. It is just easier with Cassandra so that is what we used, but you can go with any other alternative that you find convenient. We have discussed more database solutions for various scenarios in this article; you can check it out for a detailed explanation.

### Analytics

Now let’s try to improve this system a bit. Because we can extract a lot of data that can later be used for making business decisions, let's add an analytics component to our system. Every time we get a request to generate a short URL, we will get some attributes along with it, like which platform it is using - could be something like Facebook or Twitter, which user agent it is coming from - iOS, Android, web browser, etc, we will also get the IP address of the sender. These attributes can give us a lot of information like which companies are using our system - these are our clients, which part of the world most requests are coming from - we can keep an instance of the token service there to reduce latency, etc. So when we get a request, instead of straight away responding back with the longer URL, we will first save this information into a Kafka, which can be used to power the analytics. But this adds an additional step in our process, and that increases the latency. So instead of doing this in a sequential manner, we can make this write to Kafka an asynchronous parallel operation running on a separate thread.

The only challenge here is that if for some reason, the write operation fails, we don’t have that data anymore, and it will be lost. But since it is not very critical information, just the most basic information about the users, losing out on some of this data will not be a huge loss.

### Optimize Optimize Optimize

Ok, so we also have analytics built into our system now. Can we still optimize it further? Well, remember those Kafka writes after every request? We don’t necessarily need to perform them so often. Instead of sending out these events to Kafka with every request, we could maintain this information in a local data structure like with a threshold size, a queue for example, and when the data structure is about to overflow, we can do a bulk write to Kafka and empty the queue. Or we could also make it a scheduled operation, for example, do a bulk write after every 30 seconds and empty the queue. Since CPU and bandwidth utilization are reduced, we can drive more performance from the machine helping with low latency and high availability.

Again this will be an asynchronous operation, and the drawback with maintaining the data locally is that if the machine unexpectedly shuts down and the write operation fails, we will lose more information than we were losing on single writes, but you can discuss these trade-offs with your interviewer to come up with a solution that best suits your requirements.

That should be it for a URL shortening service system design!