IP2 Requirements and Architecture

Scott Zhou

I'm going to create a service that will convert a website address into a short link and sends it to the original website address. The goal is to provide a better experience for users and lower server space.

For example, if we shorten this page through TinyURL:

- https://courses.cityu.edu/ultra/courses/_153947_1/cl/outline

We would get:

- https://tinyurl.com/9z6nfucc

The shortened URL is only the half of the size of the actual URL.
URL shortening is used to improve the efficiency of linking and track individual links. It can also be used to measure ad campaigns' performance.

Requirements and Goals of the System

Functional Requirements:

1. For each page of our service, we should generate a short link. This link can be easily copied and pasted into other applications.
2. When users access a short link, our service should redirect them to the original link.
3. Users should optionally be able to pick a custom short link for their URL.
4. Links will expire after a standard default timespan. Users should be able to specify the expiration time.

Non-Functional Requirements:

1. The system should be highly available. This is required because, if our service is down, all the URL redirections will start failing.
2. URL redirection should happen in real-time with minimal latency.
3. Shortened links should not be guessable (not predictable).

Extended Requirements:

1. Analytics; e.g., how many times a redirection happened?
2. Our service should also be accessible through REST APIs by other services.

Capacity Estimation and Constraints

Our system will be read-heavy. There will be lots of redirection requests compared to new URL shortenings. Let’s assume a 100:1 ratio between read and write.

Traffic estimates: Assuming, we will have 500M new URL shortenings per month, with 100:1 read/write ratio, we can expect 50B redirections during the same period:
100 * 500M => 50B

What would be Queries Per Second (QPS) for our system? New URLs shortenings per second:
500 million / (30 days * 24 hours * 3600 seconds) = ~200 URLs/s

Considering 100:1 read/write ratio, URLs redirections per second will be:
100 * 200 URLs/s = 20K/s

Bandwidth estimates: For write requests, since we expect 200 new URLs every second, total incoming data for our service will be 100KB per second:
200 * 500 bytes = 100 KB/s

For read requests, since every second we expect ~20K URLs redirections, total outgoing data for our service would be 10MB per second:
20K * 500 bytes = ~10 MB/s

Memory estimates: If we want to cache some of the hot URLs that are frequently accessed, how much memory will we need to store them? If we follow the 80-20 rule, meaning 20% of URLs generate 80% of traffic, we would like to cache these 20% hot URLs.

Since we have 20K requests per second, we will be getting 1.7 billion requests per day:
20K * 3600 seconds * 24 hours = ~1.7 billion

To cache 20% of these requests, we will need 170GB of memory.
0.2 * 1.7 billion * 500 bytes = ~170GB

One thing to note here is that since there will be many duplicate requests (of the same URL), our actual memory usage will be less than 170GB.

Database Design

A few observations about the nature of the data we will store:
1. We need to store billions of records.
2. Each object we store is small (less than 1K).
3. There are no relationships between records—other than storing which user created a URL.
4. Our service is read-heavy.

Database Schema:
We would need two tables: one for storing information about the URL mappings and one for the user's data who created the short link.

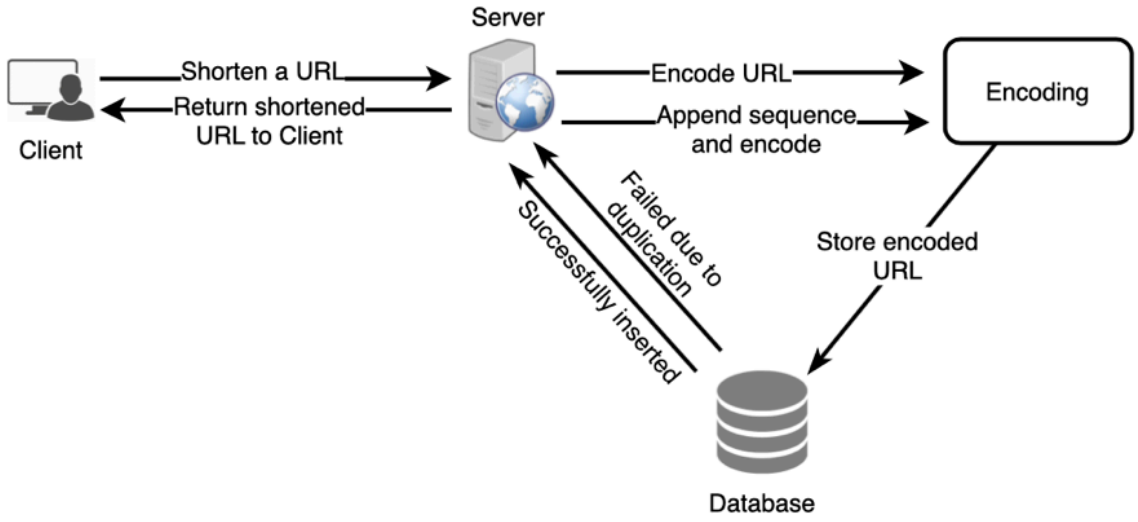
URL: PK Hash:varchar(16), OrginalURL: varchar(512), CreationDate:datetime, ExpirationDate:datetime, UserID: int.

User: PK UserID:int, Name: varchar(30), Email: varchar(32), CreationDate: datetime, LastLogin: datetime.

Basic System Design and Algorithm

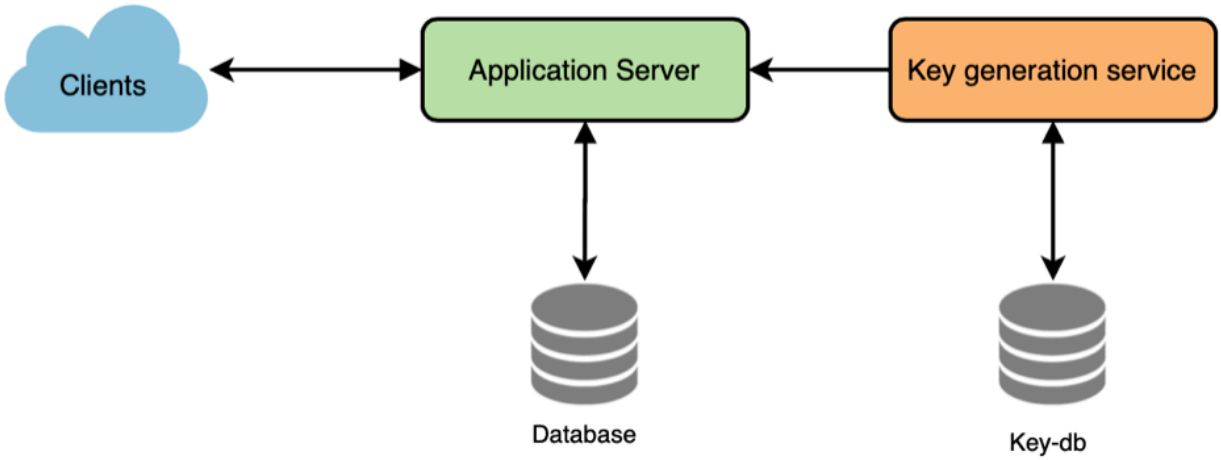
The problem we are solving here is how to generate a short and unique key for a given URL.
In the TinyURL example in previous slide. the shortened URL is "https://tinyurl.com/9z6nfucc". The last seven characters of this URL is the short key we want to generate.
We'll explore two solutions here:

We can compute a unique hash of the give URL by using MD5 or SHA256. The Base64 encoding could give between 68 billion to 281 billion possible strings.



The MD5 will produce a 128-bit hash value.

Second solution is using standalone Key Generation Service which will randomly generate 6 letters strings beforehand and put them in the database. Every time the URL generation happen we can just take the already generated keys and use it.



Reference:

IP2 Requirements and Architecture

Scott Zhou

I'm going to create a service that will convert a website address into a short link and sends it to the original website address. The goal is to provide a better experience for users and lower server space.

For example, if we shorten this page through TinyURL:

- https://courses.cityu.edu/ultra/courses/_153947_1/cl/outline

We would get:

- https://tinyurl.com/9z6nfucc

The shortened URL is only the half of the size of the actual URL.
URL shortening is used to improve the efficiency of linking and track individual links. It can also be used to measure ad campaigns' performance.

Requirements and Goals of the System

Functional Requirements:

1. For each page of our service, we should generate a short link. This link can be easily copied and pasted into other applications.

2. When users access a short link, our service should redirect them to the original link.
3. Users should optionally be able to pick a custom short link for their URL.
4. Links will expire after a standard default timespan. Users should be able to specify the expiration time.

Non-Functional Requirements:

1. The system should be highly available. This is required because, if our service is down, all the URL redirections will start failing.
2. URL redirection should happen in real-time with minimal latency.
3. Shortened links should not be guessable (not predictable).

Extended Requirements:

1. Analytics; e.g., how many times a redirection happened?
2. Our service should also be accessible through REST APIs by other services.

Capacity Estimation and Constraints

Our system will be read-heavy. There will be lots of redirection requests compared to new URL shortenings. Let's assume a 100:1 ratio between read and write.

Traffic estimates: Assuming, we will have 500M new URL shortenings per month, with 100:1 read/write ratio, we can expect 50B redirections during the same period:
100 * 500M => 50B

What would be Queries Per Second (QPS) for our system? New URLs shortenings per second:
500 million / (30 days * 24 hours * 3600 seconds) = ~200 URLs/s

Considering 100:1 read/write ratio, URLs redirections per second will be:
100 * 200 URLs/s = 20K/s

Bandwidth estimates: For write requests, since we expect 200 new URLs every second, total incoming data for our service will be 100KB per second:
200 * 500 bytes = 100 KB/s

For read requests, since every second we expect ~20K URLs redirections, total outgoing data for our service would be 10MB per second:
20K * 500 bytes = ~10 MB/s

Memory estimates: If we want to cache some of the hot URLs that are frequently accessed, how much memory will we need to store them? If we follow the 80-20 rule, meaning 20% of URLs generate 80% of traffic, we would like to cache these 20% hot URLs.

Since we have 20K requests per second, we will be getting 1.7 billion requests per day:
20K * 3600 seconds * 24 hours = ~1.7 billion

To cache 20% of these requests, we will need 170GB of memory.
0.2 * 1.7 billion * 500 bytes = ~170GB

One thing to note here is that since there will be many duplicate requests (of the same URL), our actual memory usage will be less than 170GB.

Database Design

A few observations about the nature of the data we will store:
1. We need to store billions of records.
2. Each object we store is small (less than 1K).
3. There are no relationships between records—other than storing which user created a URL.
4. Our service is read-heavy.

Database Schema:
We would need two tables: one for storing information about the URL mappings and one for the user's data who created the short link.

URL: PK Hash:varchar(16), OrginalURL: varchar(512), CreationDate:datetime, ExpirationDate:datetime, UserID: int.

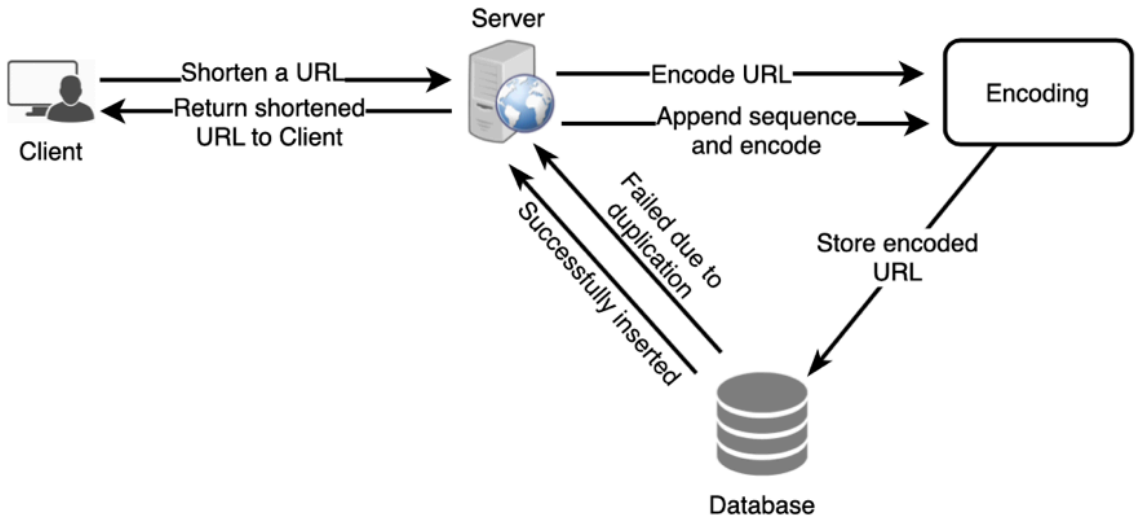
User: PK UserID:int, Name: varchar(30), Email: varchar(32), CreationDate: datetime, LastLogin: datetime.

Basic System Design and Algorithm

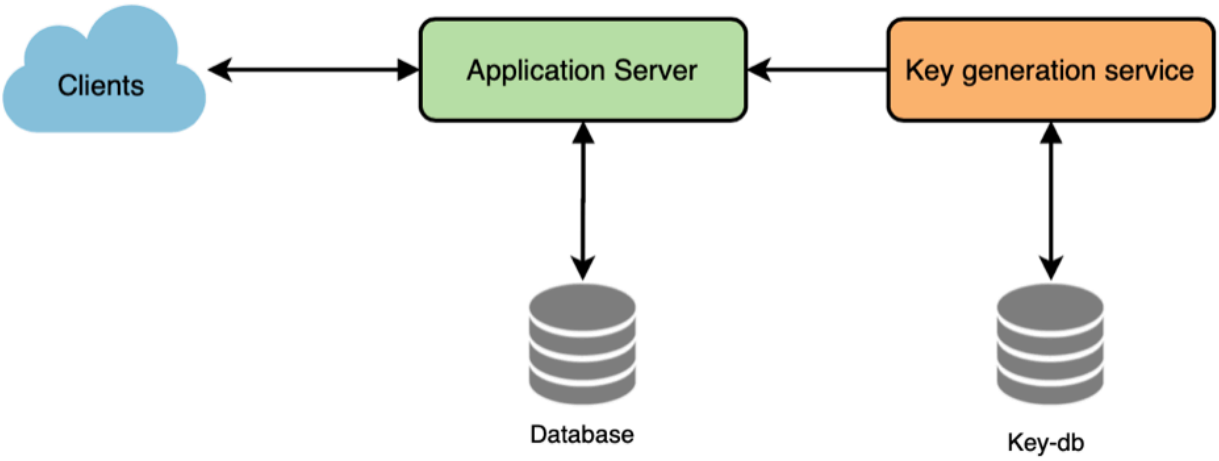
The problem we are solving here is how to generate a short and unique key for a given URL.
In the TinyURL example in previous slide. the shortened URL is "https://tinyurl.com/9z6nfucc". The last seven characters of this URL is the short key we want to generate.
We'll explore two solutions here:

We can compute a unique hash of the give URL by using MD5 or SHA256. The Base64 encoding could give between 68 billion to 281 billion possible strings.

The MD5 will produce a 128-bit hash value.



Second solution is using standalone Key Generation Service which will randomly generate 6 letters strings beforehand and put them in the database. Every time the URL generation happen we can just take the already generated keys and use it.



Reference:

Grokking the System Design Interview. (n.d.). Educative. https://www.educative.io/courses/grokking-the-system-design-interview