

Designing YouTube

Difficulty Level: Medium

Let's design a video sharing service like YouTube, where users will be able to upload, view, and search videos.

1. Why YouTube?

YouTube is one of the most popular video sharing websites in the world.

Users of the service can upload, view, share, rate, and report videos, as well as add comments on videos.

2. Requirements and Goals of the System

For the sake of this exercise, we plan to design a simpler version of YouTube with the following requirements:

Functional Requirements

1. Users should be able to upload videos
2. Users should be able to share and view videos
3. Users should be able to perform searches based on video titles
4. Our services should be able to record stats of videos, e.g. likes, dislikes, total views etc.
5. Users should be able to add and view comments on videos

Non-Functional Requirements

1. The system should be highly reliable, any video uploaded should not be lost
2. The system should be highly available — consistency can take a hit (in the interest of availability), so if a user doesn't see a video for a while, this is fine
3. Users should have a real-time experiences while watching videos and should not feel any lag

Not in scope: video recommendations, most popular videos, channels, subscriptions, watch later, favourites etc.

3. Capacity Estimation and Constraints

Let's assume that we have 1.5 billion total users, 800 million of whom are DAU.

If, on average, a user views 5 videos per day, then the total video-views per second would be:

$$800M * 5 / 86400 \text{ sec} = 46K \text{ views/sec}$$

Let's assume that our upload:view ratio is 1:200, i.e. for every video upload we have 200 videos viewed. This gives us 230 videos uploaded per second:

$$46K / 200 = 230 \text{ uploads/sec}$$

Storage Estimates: Let's assume that every minute 500 hours worth of videos are uploaded to YouTube. If, on average, one minute of video needs 50MB of storage (videos need to be stored in multiple formats), then the total storage needed for videos uploaded in a minute would be:

$$500 \text{ hrs} \cdot 60 \text{ mins} \cdot 50 \text{ MB} = 1500 \text{ GB/min} = 25 \text{ GB/sec}$$

These are estimated numbers ignoring video compression and replication, which would change real numbers.

Bandwidth Estimates: With 500 hours of video uploads per minute (which is 30,000 mins of video uploads per minute), assuming that each minute of the video takes 10MB of the bandwidth, we would be getting 300GB of uploads every minute:

$$500 \text{ hrs} \cdot 60 \text{ mins} \cdot 10 \text{ MB} = 300 \text{ GB/min} = 5 \text{ GB/sec}$$

Assuming an upload:view ratio of 1:200, we would need 1TB/s of outgoing bandwidth.

In summary, 5GB/sec of ingress, and 1TB/s of egress.

4. System APIs

We can have SOAP or REST APIs to expose the functionality of our service.

The following could be the definition of the API for uploading videos:

```
uploadVideo(api_dev_key, video_title, video_description, tags[], category_id, default_language, recording_details, video_contents)
```

Parameters:

- `api_dev_key` (string) — the API developer key of a registered account; this will be used to, among other things, throttle users based on their allocated quota
- `video_title` (string) — title of the video
- `video_description` (string) — optional description of the video
- `tags` (string[]) — optional tags for the video
- `category_id` (string) — category of the video, e.g. film, song, gaming etc.
- `default_language` (string) — e.g. English, Mandarin, Malay etc.
- `recording_details` (string) — location where the video was recorded
- `video_contents` (fstream) — video to be uploaded

Returns: (string)

A successful upload will return a HTTP 202 (request accepted) response, and once the video encoding is completed, the user is notified by email with a link to access the video. Otherwise, a HTTP 406 (not acceptable) response is returned.

We can also expose a queryable API to let users know the current status of their uploaded video.

The following could be the definition of the API for searching for a video:

```
searchVideo(api_dev_key, search_query, user_location, maximum_videos_to_return, page_token)
```

Parameters:

- `api_dev_key` (string) — same as above
- `search_query` (string) — a string containing the search terms
- `user_location` (string) — optional location of the user performing the search

- `maximum_videos_to_return` (number) — maximum number of results to be returned in one request
- `page_token` (string) — this token will specify a page in the result set that should be returned

Returns: (JSON)

A JSON containing information about the list of video resources that match the search query.

Each video resource will have a video title, a thumbnail, a video creation date, and a view count.

The following could be the definition of the API for streaming a video:

```
streamVideo(api_dev_key, video_id, offset, codec, resolution)
```

Parameters:

- `api_dev_key` (string) — same as above
- `video_id` (string) — a string to identify the video
- `offset` (number) — we should be able to stream the video from any offset; this offset would be a time in second from the beginning of the video.

If we support playing/pausing a video from multiple devices, we will need to store the offset on the server. This will enable the users to start watching a video on any device from the same point where they left off.

- `codec` (string) and `resolution` (string) — we should send the codec and resolution info in the API from the client to support play/pause from multiple devices.

E.g. imagine you are watching a video on your TV app, paused it, and then started watching it on your phone app. In this case, you would need codec and resolution, as both of these devices have a different resolution and use a different codec.

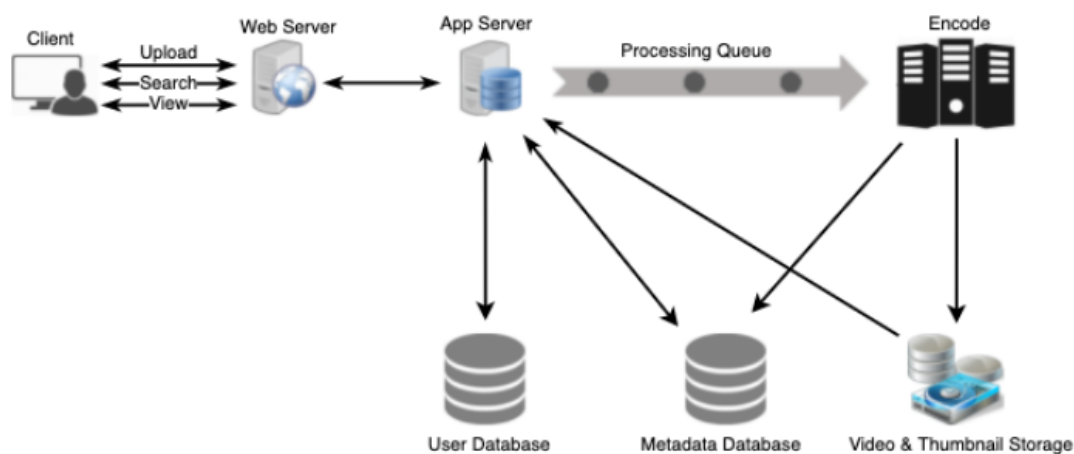
Returns: (fstream)

A media stream (a video chunk) will be returned from the given offset on success. Otherwise, an error will be returned.

5. High-Level Design

At a high-level, we would need the following components:

1. **Processing Queue** — each uploaded video will be pushed to a processing queue to be dequeued later for encoding, thumbnail generation, and storage
2. **Encoder** — to encode each uploaded video into multiple formats
3. **Thumbnails Generator** — to generate a few thumbnails for each video
4. **Video and Thumbnail Storage** — to store video and thumbnail files in some distributed file storage
5. **User Database** — to store user's information, e.g. name, email address etc.
6. **Video Metadata Storage** — a metadata database to store all the information about videos like title, file path in the system, uploading user, total views, likes, dislikes etc.; we also use it to store the video comments



High level design of Youtube

6. Database Schema

Video Metadata Storage

Video metadata can be stored in a SQL database, e.g. MySQL or PostgreSQL.

In our Video table, we will store the following information in every record:

- VideoID
- Title
- Description
- Size
- Thumbnail
- Uploader/User
- Total number of likes
- Total number of dislikes
- Total number of views

In our Comment table, we will store the following information in every record:

- CommentID
- VideoID
- UserID
- Comment
- Time of creation

User Data Storage

We can also use a SQL database to store our user data.

In our User table, we will store the following information in every record:

- UserID,
- Name,
- Email address
- Age
- Registration details
- Time of creation
- Etc

7. Detailed Component Design

The service would be read-heavy, so we will focus on building a system that can retrieve videos quickly.

We can expect our read:write ratio to be 200:1, meaning that for every video upload, there are 200 video views.

Where will videos be stored?

Videos can be stored in a distributed file storage system like HDFS or GlusterFS.

How should we efficiently manage read traffic?

We should segregate our read traffic from our write traffic.

Since we will have multiple copies of each video, we can distribute our read traffic towards different servers.

For metadata, we can have primary-secondary configurations, where writes will go to the primary first, and then be replicated onto the secondaries.

Such configurations can cause some staleness in data — e.g. when a new video is added, its metadata will be inserted into the primary, and will not be visible to the secondaries until the data is replicated from the primary to the secondaries, therefore the secondaries return stale results to the user.

This staleness might be acceptable in our system as it would be very short-lived, and the user would be able to see the new videos after a few milliseconds.

Where would thumbnails be stored?

If we assume that every video has five thumbnails, we get the situation where there will be a lot more thumbnails than videos.

Hence, we need to have a very efficient storage system that can server huge read traffic.

There will be two considerations before deciding which storage system should be used for the thumbnails:

1. Thumbnails are small files, say, a maximum of 5KB each

2. Read traffic for thumbnails will be huge compared to videos — users will be watching one video at a time, but they might be looking at a page with 20 thumbnails of other videos

If we store the thumbnails on the disk, we will have to perform many seeks to different locations on the disk to read our huge number of files.

Thus, storing on disk is quite inefficient and will result in high latencies.

Bigtable can be a reasonable choice here, as it combines multiple files into one block to store on the disk, and is very efficient in reading a small amount of data.

Both of these are the two most significant requirements for our service.

Keeping hot thumbnails in a cache will also help improve the latencies (we can use the 80-20 rule), and given that the thumbnail files are small in size, we can easily cache a large number of such files in memory.

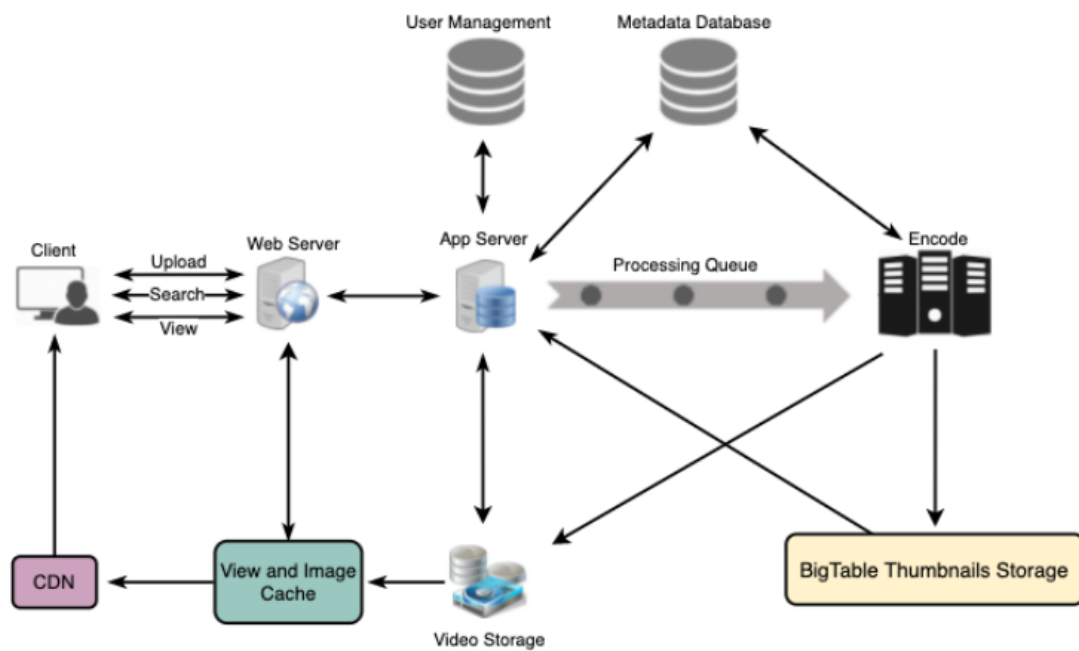
Video Uploads

Since video uploads could be huge, if while uploading, the connection drops, we should support resuming from the same point.

Video Encoding

Newly uploaded videos are stored on the server, and a new task is added to the processing queue to encode the video into multiple formats.

Once all the encoding is completed, the uploaded will be notified, and the video is made available for view/sharing.



Detailed component design of Youtube

8. Metadata Sharding

Since we have a huge number of new videos every day, and our read load is extremely high, we need to distribute our data onto multiple machines so that we can perform read-write operations efficiently.

We have many options to shard our data. Let's go through the different options one-by-one.

Sharding based on UserID

We can try storing all the data for a particular user on a single server.

While storing, we can pass the UserID to our hash function, which will map the user to a database server that will store all of the metadata for that user's videos.

While querying for the videos of a user, we can ask our hash function to find the server holding the user's data, and then read it from there.

To search videos by title, we will have to query all servers, and each server will return a set of videos. A centralised server will then aggregate and rank these results before returning them to the user.

This approach has a couple of issues:

1. What if a user becomes popular? There could be a lot of queries on the server that holds that user, creating a performance bottleneck. This will also affect the overall performance of our service.
2. Over time, some users can end up storing a lot of videos compared to others. Maintaining a uniform distribution of growing user data is tricky.

To recover from these situations, either we have to repartition or redistribute our data, or use consistent hashing to balance the load between servers.

Sharding based on VideoID

Our hash function will map each VideoID to a random server where we will store that video's metadata.

To find the videos of a user, we will query all servers, and each server will return a set of videos. A centralised server will aggregate and rank these results before returning them to the user.

This approach solves our problem of popular users, but shifts it to popular videos.

We can further improve our performance by introducing a cache to store hot videos in front of the database servers.

9. Video Deduplication

With a huge number of users uploading a massive amount of video data, our service will have to deal with widespread video duplication.

Duplicate videos often differ in aspect ratios or encodings, contain overlays or additional borders, or be excerpts from a longer original video.

The proliferation of duplicate videos can have an impact on many levels:

1. Data storage — we could be wasting storage space by keeping multiple copies of the same video
2. Caching — duplicate videos would result in degraded cache efficiency by taking up space that could be used for unique content

3. Network usage — duplicate videos will also increase the amount of data that must be sent over the network to in-network caching systems
4. Energy consumption — higher storage, inefficient cache, and higher network usage could result in energy wastage

For the end-user, these inefficiencies will be realised in the form of duplicate search results, longer video start-up times, and interrupted streaming.

For our service, deduplication makes the most sense early — when a user is uploading a video as compared to when it is being post-processed to later find duplicate videos.

Inline deduplication will save us a lot of resources that can be used to encode, transfer, and store the duplicate copy of the video.

As soon as any user starts uploading a video, our service can run a video matching algorithm (e.g. Block Matching, Phase Correlation etc.) to find duplications.

If we already have a copy of the video being uploaded, we can either stop the upload and use the existing copy, or continue the upload and sue the newly uploaded video if it is of higher quality.

If the newly uploaded video is a subpart of an existing video or vice versa, we can intelligently divide the video into smaller chunks so that we only upload the parts that are missing.

10. Load Balancing

We should use consistent hashing among our cache servers, which will also help in balancing the load between cache servers.

Since we will be using a static hash-based scheme to map videos to hostnames, there may be an uneven load on the logical replicas due to each video's differing popularity.

For instance, if a video becomes popular, the logical replica corresponding to that video will experience more traffic than other servers.

These uneven loads for logical replicas can then translate into uneven load distribution on their corresponding physical servers.

To resolve this issue, any busy server in one location can redirect a client to a less busy server in the same cache location. We can use dynamic HTTP redirections for

this scenario.

However, the use of redirections also has its drawbacks.

First, since our service tries to load balance locally, it leads to multiple redirections if the host that receives the redirection can't serve the video.

Also, since each redirection requires a client to make an additional HTTP request, it also leads to higher delays before the video starts playing back.

Moreover, inter-tier (or cross data center) redirections lead a client to a distant cache location because the higher tier caches are only present at a small number of locations.

11. Cache

To serve globally distributed users, our service needs a massive-scale video delivery system.

Our service should push its content closer to the user using a large number of geographically distributed video cache servers.

We need to have a strategy that will maximise user performance, and also evenly distributes the load on its cache servers.

We can introduce a cache for metadata servers to cache hot database rows.

We can use Memcached to achieve this.

Applications servers will check for a cache hit before going to the database.

Least Recently Used (LRU) can be a reasonable cache eviction policy for our system.

How can we build a more intelligent cache?

If we go with the 80-20 rule, i.e. 20% of daily read volume for videos is generating 80% of the read traffic, meaning that certain videos are so popular that the majority of people are viewing them, it follows that we can try caching 20% of the daily read volume of videos and metadata.

12. Content Delivery Network (CDN)

A CDN is a system of distributed servers that deliver web content to a user based on their geographic location, the origin of the web page, and a content delivery server.

Our service can move popular videos to CDNs:

- CDNs replicate content in multiple places — there is a better chance of videos being closer to the user and, with fewer hops, videos will stream from a closer and friendlier network
- CDN machines make heavy use of caching, and can mostly server videos out of memory.

Less popular videos (1-20 views per day) that are not cached by CDNs can be served by our servers in various data centers.

13. Fault Tolerance

We should use consistent hashing for distribution among database servers.

Consistent hashing will not only help in replacing a dead server, but also help in distributing load among servers.