eileen-code4fun.medium.com

System Design Interview: Mini YouTube - Eileen Pangu - Medium

Eileen Pangu

8-10 minutes

Design a video service like YouTube, or Netflix. I don't know if companies really ask this question, but it's in a lot of system design materials. I skimmed through many, most of them are not as technical as I'd like. So I decide to write my own, and I hope that it offers a different view to you.

As always, let's clarify what we are trying to design here. YouTube is so broad. But at the very minimum, we should expect content creators to be able to upload their videos; viewers can find the videos they are looking for, and then binge on them. Those are the fundamental functions of a video service site. One can have all kinds of fancy features such as recommendation, subscription, ads, community engagement, etc. But without a solid foundation, everything will eventually fall apart. So, let's restate, our goals are easy upload, flexible search, and fast reviewing. Let's go through them one by one.

Video Upload

Video upload is relatively straightforward, assuming that users have some kind of client (most likely a frontend app in browser)

that can talk to the server. The client just needs to read the local video file and upload that to the server. YouTube has many supported video formats, and a set of recommended encoding settings. The encoding jargon seems quite daunting. Let's explain them a bit. The container format is the video file format (e.g. MP4). It's a wrapper around the actual content to provide some additional information. The codec is a shorthand for encoder and decoding. It describes how the digital signal of the video and audio should be compressed for storage/transmission and decompressed for playing. The framerate is the number of frames per second. A frame is one still image in the video. The resolution is self-explanatory. The bitrate is the number of information bits per second when you play the video. The bitrate of a raw video is just the resolution times the framerate times the size of a pixel. But usually the video is compressed, and the bitrate is capturing the bits per second after compression, which varies due to different compression ratios. Even though these details won't affect the upload process, they are helpful context to keep in mind for any video site.

The actual upload usually starts with an HTTP request to the server. The client probably won't start uploading the video in this first request, which is instead commonly used to establish various settings, such as creating a video ID, preparing a storage location, so on and so forth in the server. Once the server responds the storage location to upload the video, the client starts reading the video file as binary and send it over to the server.

Our goal is easy upload. One conspicuously missing feature in the current design is that the upload is not resumable. Many content creators may be on the road. They may not necessarily always

have a bulk chunk of time with stable network connection to upload the entire video file all at once. So we need to build to tolerate network interruption. This requires coordination between the client and the server. The client needs to send the files in chunks and communicate with the server to confirm accepted chunks. After network interruption and the client comes back online, it can ask the server which chunk was last received. Once the client knows that, it can calculate corresponding offset in the video file to resume the upload. See an overview in Figure 1.

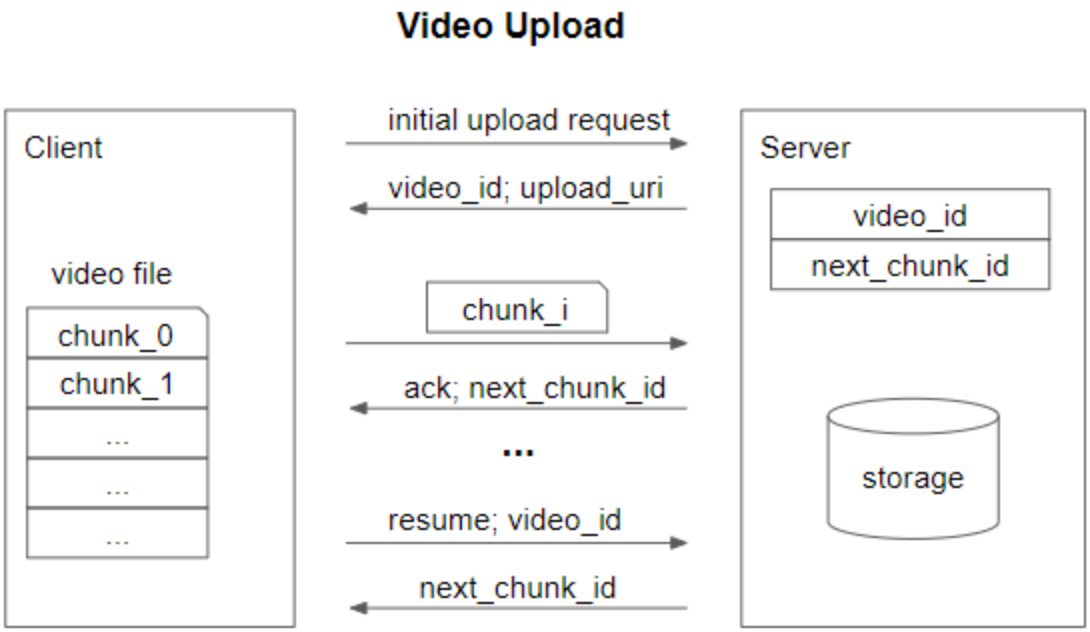

Figure 1

The server collects metadata about the uploaded video, things like title, timestamp, language, category, tags, etc, which are useful attributes for search.

Search

Viewers want to search for videos. During the upload, the video information is saved in a video_info table in the server's database. It contains the video ID, upload timestamp, owner, and metadata

such as title, language, category, tags, etc. The actual video content is usually not stored in the video_info table but rather in some blob store and a reference URI is saved in the database. See Figure 2 for an illustration.

vid	upload_ts	owner_email	metadata	storage_uri
123	2020-01-01 12:00:03 GMT	foo@example.com	{ title: ... language: ... tags: ... category: ... }	ss://a/b/c/123

Figure 2

We will then build a reverse search index that maps keywords to video ID. The keywords are obtained from the video_info attributes. When viewers search for the keywords, the server will locate and respond with the relevant video IDs. For a more in-depth indexing and search discussion, I have a [mini Google system design post] that you can check out.

Video Streaming

Now comes the most exciting part — video streaming. Before anything, the client player needs to be compatible with the video format so that it can actually understand the video. This is why I mentioned the encoding settings in the upload section. Let's assume that the client player (most often the browser) supports the necessary encoding settings, then it's just a matter of delivering the data from the server to the client.

The first meaty discussion point is the streaming protocol. Even though much of the world wide web is built on TCP, video

streaming actually heavily replies on UDP. The primary reason is that video streaming is performance/latency sensitive and can tolerate network packet loss. Imagine a packet is delayed in the network transmission. TCP has to wait for it because it guarantees ordering. It may sometimes need to ask the sender to resend the packet because it guarantees delivery. This will create a noticeable delay in the video streaming, and viewers have to wait longer. And because TCP guarantees so much, it's generally slower. UDP, on the order hand, doesn't provide those guarantees. So it's faster. In the case of packet loss, the client can simply discard it and move on. Viewers may see a small blurry area in a frame or two, but that's generally better than having to constantly wait for the video to load. UDP also supports multicast, enabling the server to stream to multiple viewers simultaneously. In this case, the server only sends out one copy of the data. It's the network elements (routers, switches) along the way that duplicate the packets. This drastically saves the server's bandwidth, and it's particularly useful in live streaming scenarios.

Now, viewers can watch the entire video. What if they want to seek to a particular point that's not buffered? To support this, the server has to do more preprocessing work. It needs to split (virtually or physically) the video into chunks. Note that the chunk boundary should usually retain complete frames. And the server needs to build a mapping from video play time to chunks. When the client asks for a specific video play time, the server can stream from the nearest chunks. This is also a useful technique to resume playing. Obviously this requires the client to obtain the video metadata from the server so that it knows the total play time. Note that the chunk for playing is usually much granular than the chunk for uploading.

This is because we want to be precise in response to the seek. See Figure 3 for an all-in-one overview.

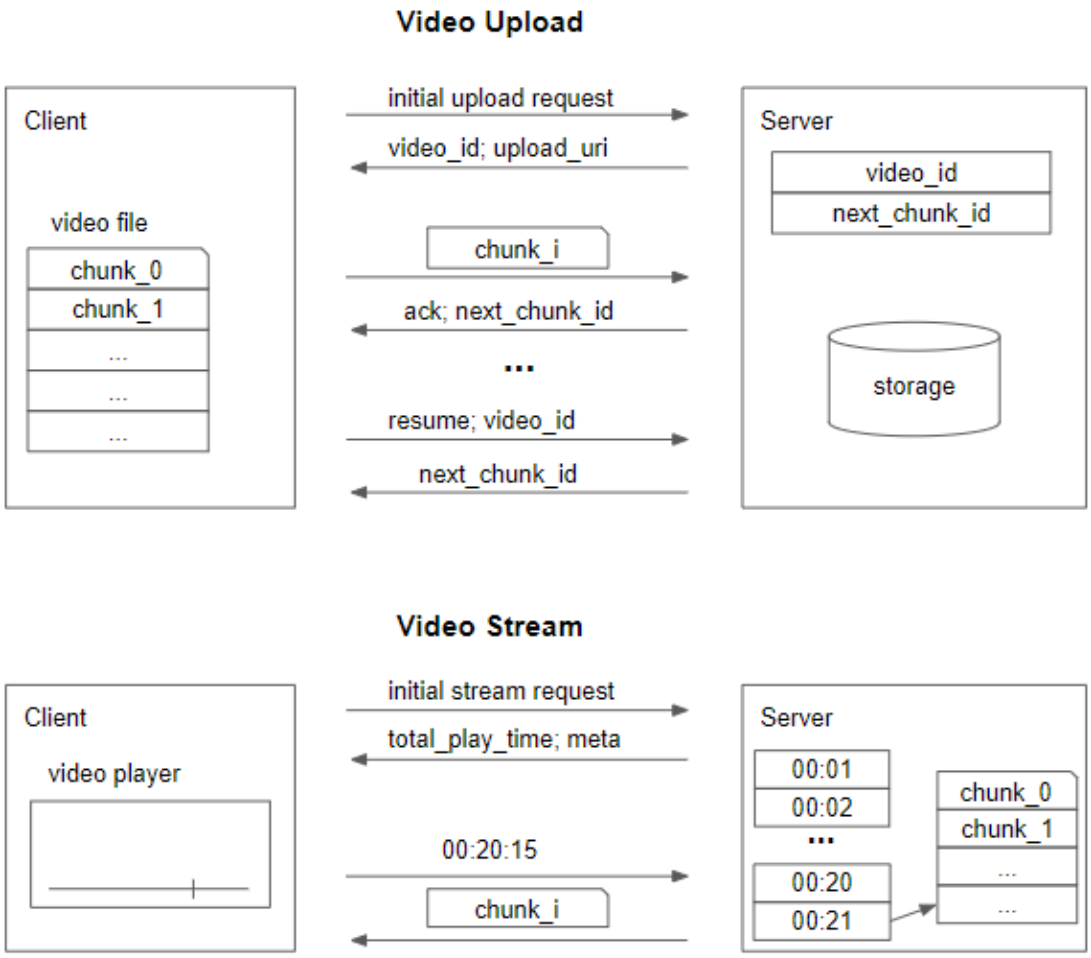

Figure 3

All is good now. But how do we scale up our video streaming? The answer used to be “content delivery network” (CDN). CDN is mainly used to scale across geolocations and reduce latency. Instead of building data centers and servers across the globe, we can tap into the existing CDNs which already have global footprints. The video contents will be copied and stored by servers inside CDN across geolocations. We also have to modify the DNS records of our service domain so that video requests are routed to CDN servers in their vicinity. CDN is as much a technology as it’s a business model. It’s a form of shared economy.

The reason why I said it “used to be” is because I think we have a prominent alternative now. If you think about it, the essential value CDN provides is that we don’t have to build data centers and servers across geolocations. Cloud computing platform is doing exactly that and in a more generic way. We can replicate content across the globe and deploy servers in multiple regions on cloud. And we’ll only be charged for what we consume. Yes, that’s a lot of maintenance hassles. But what we gain is the full control of the stack. We don’t have to share content with other organizations (CDN service providers). User requests go directly to our own servers, where we can track all kinds of metrics. We own the entire network and can keep a closer eye on anything that goes off the track.

Closing

I think this is the end of the system design interview. YouTube is obviously much more complex than this. Even a basic video sharing/viewing site has much more to consider. But I hope that I’ve scoped out the fundamental components, and that they make sense to you.