

Twitch

Problem Requirements

- 1) Streamers can send their footage to many clients around the world
- 2) Viewers can adaptively change stream quality as their network requires without dropping frames
- 3) We can record streams and play them back later
- 4) Stream chat

Scale: streams can have hundreds of thousands of concurrent viewers

Networking

Recall: In our zoom video, we used UDP

→ Faster

→ We don't care about dropped frames, by the time they arrive it's too late

Is this still the case? We want to be able to have high quality streams

→ If streams are instantly sent to users, resending frames is useless

→ If we add a few seconds of delay to our stream...

1) There is enough time to re-send dropped frames to the user!

2) We can use TCP!

→ This is the premise behind RTMP (real time messaging protocol)

you won't notice a few seconds of delay of the stream

Data Chunking

It may be greatly beneficial to us to have our streamer send their video data in chunks!

- Fewer messages to send than per frame
- Fewer messages for clients to receive
- Fewer video data metadata rows to persist
 - For watching back stream later
 - For aligning timestamps of chunks of different resolutions for slow clients











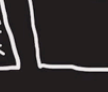

Note: We don't want to make chunks too big!

- Chunks that fail to send have to be re-sent, can be expensive!

if we have bad wifi or our slow our resolution has to decrease by itself and we can still continue seeing the video

Aligning Chunks For Slow Clients

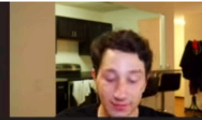
We saw this same technique when building YouTube!

Resolution:	Timestamp: <u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>
<u>480p</u>				
<u>720p</u>				
<u>1080p</u>				

As the viewer's network speed slows down/speeds up, we can request the chunk at the next timestamp with a more suitable resolution!

this is also a reason why it is better to use standardized chunk sizes

Delivering Video To Viewers



Should clients poll the streamer PC directly for video footage? Probably not.

→ There can be lots of viewers

→ We want to minimize load on the streamer's setup to keep stream quality high

We can introduce a server that the streamer publishes to!

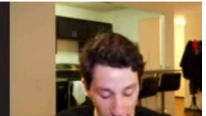
Should the streamer publish multiple resolutions of their own stream?

→ Unlike Zoom, server only gets one footage source per stream

→ We want to keep load as low as possible on the streamer PC

Let's do encoding server side!

Replication/Partitioning of Stream Server



How should we partition our stream servers?

→ Using something like IP would be bad if streamer switches devices/networks

→ Let's do consistent hashing on stream Id!

What about fault tolerance?

→ We have two options!

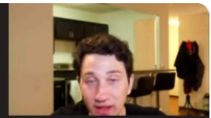
→ On stream server failure, use consistent hashing to evenly distribute streamers to other stream servers

→ Could overload existing stream servers

→ On node failure redirect all of its traffic to a passive backup

→ Seems better

Loading Streams Faster



In our current setup, we will have trouble with:

- Streams with lots of users
- Streams with viewers all over the globe

How can we improve this? Caching!

- CDNs for video chunks and caches for chunk metadata!



Push vs. Pull Caching

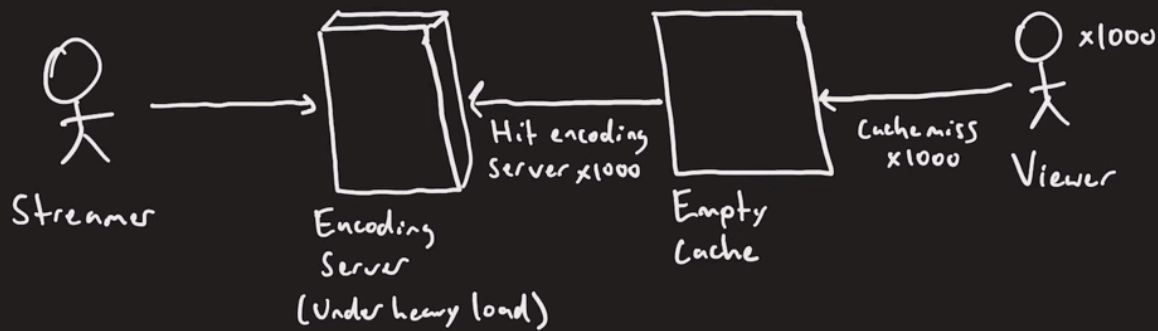
It would be great if our server could push right to caches in order to avoid any initial caches!

- Do we always know when streams will be popular in advance?
- Verified users
 - Pushable
- Unpopular users via hosting / a stream raid
 - In this case we don't know in advance they'll get lots of viewers
 - Will likely have to use a pull based cache

Using pull based caching can lead to problems!

Thundering Herd

Imagine a situation where an unpopular stream suddenly gets lots of views



This thundering herd can be avoided by using a lock!

→ Cache should only allow one request through to the server at a time

when the first request comes it will grab a lock of the cache and be let through to the encoding server, the other requests will have to wait as they don't have a lock. finally the first request will go through and release the lock and now the other requests will hit the cache, since now it has already been cached from the first request it will be much faster for the other requests

Stream Recording

To be able to ensure that we have access to streams later, we need to offload the data from the server!

We want:

- Fast ingestion of metadata
- Fast reads of metadata
- Lots of storage for video chunks (S3)

Let's use HBase for metadata!

- LSM tree for fast writes
- Column oriented for fast reads
- Partition by streamId, sort by timestamp

Stream Chat

Note: Clients just have one chat they care about at a time

Easy solution:

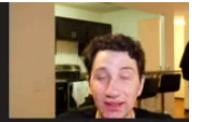
- Everyone connects to chat server based on stream Id (use consistent hashing)
- Use websockets to send messages in real time

If we have too many viewers for a given stream this breaks down!

- Too many websocket connections for the single server
- Too many messages to deliver to each client

What can we do instead?

Popular Stream Chat



Idea: All messages sent to a database, every few seconds each client pulls in the last 10 messages (if they scroll up they'll load the 10 before that)

→ We need a database with really fast ingestion speed!

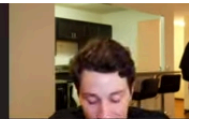
→ Cassandra! LSM trees plus leaderless replications

→ While leaderless replication probably isn't causally consistent, who cares it's twitch chat

→ Partition on streamId, sort on message timestamp (not perfect but who cares)

Now each client can read messages at its own pace!

When chat server sees that it has too many websocket connections, it can say to ZooKeeper that messages for this chat can be pulled



Twitch.tv

