Problem Statement:
Live video broadcasting app

Requirements:

- Users should be able to see a **list of live stream events**.
- Users should be able to **stream videos** at low latency from any part of the world.
- Users should be able to have **no/minimal buffering** during the video from any part of the world.
- System should be able to serve video in low/high quality formats based on consumer's demand and network quality(**Adaptive bit rate**).
- Broadcasters should be able to **broadcast videos seamlessly** without loss of content from any part of the world.
- Users should be able to watch videos of live streams they have missed after the stream has completed.

Prioritized Requirements:

- Broadcasters streaming video
- Users watching the live stream seamlessly
- Adaptive bit rate should be effective
- Fault tolerance for outrages and system breakdowns.

Requirements not part of below design:

- User authentication and session management
- Role based access control to ensure only broadcasters can publish videos.
- Live commenting/emoji reactions on live streams
- Payments and user subscriptions
- Users should be able to see completed streams later on after the broadcast is completed.

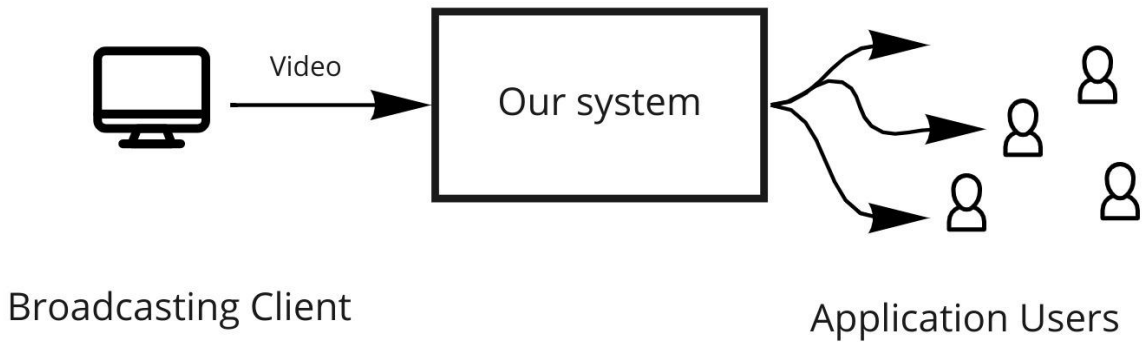
Estimated Scale of the system:

- Multiple streams could go on in parallel. To keep it simple for now let's consider a maximum of 3 events can happen in parallel (unlike youtube or twitch streams where mannier streamers would be producing content at once) .
- Millions of users will be concurrently watching a stream.
- Most of the viewers will be from a similar region though not necessary. IPL stream might have maximum audience from India and a NBA stream could have most of the audience from the west.

Expected disasters :

- Bad internet at the end of the broadcasting client.
- Outrages during the stream

Overview of a single live stream



Distributed Design Trade offs :

- Since we want the system to be highly available we can compromise on Consistency and go with **High availability and partitioning tolerance** according to CAP Theorem and make the system **eventually consistent**.

Design Challenges:

1. If we had to do a video stream where we have the full video beforehand we could simply store it in a distributed storage and serve it over a cdn.
2. We cannot ensure the quality of video being sent by the broadcaster. Also we cannot assume the format and encoding to be as we expect.

Architecture: When video is being broadcasted by one of the clients

This is the scenario where video is being published by the client. Let’s assume it's an IPL match going on in Bangalore.

- Micro service based architecture where each service has a designated responsibility.
- Content distributed across regions for fast accessibility.

Architecture Diagram

Components:

Gateway Load balancer:

1. Responsible for balancing the load across instances.
2. A uuid to be sent in the request by client so we can use consistent hashing and forward all the requests coming from a single client to a single pod.

Video Processing Micro Service:

1. Responsible for processing incoming video streams.
2. Invokes an encoder worker node to encode the incoming video to all the desired formats.
3. Chunks the videos and writes it to a big cache storage.
4. Needs to be vertically scalable. Since the whole video has to be processed continuously ideally. As per our assumption there will be fewer simultaneous broadcasts hence we need not need too many of these instances.
5. Pushes a event into the message broker when a new chunk has began to process/or processing is completed

Big cache storage:

1. Blob store responsible for storing video data

Message Broker:

1. Responsible for collecting events from the cluster.

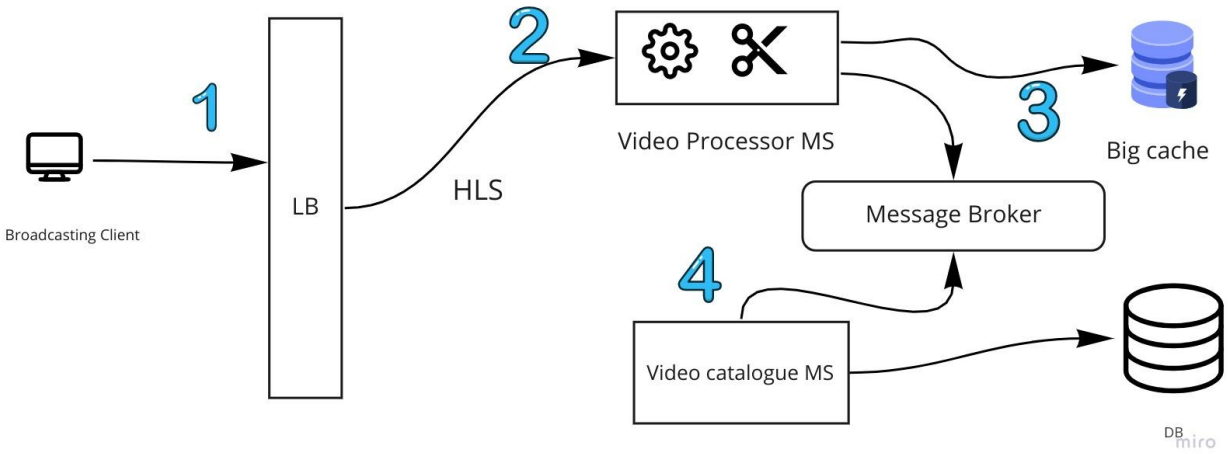
Video Catalogue Micro service:

1. Stores and processes metadata of a video, Whenever an event corresponding to a new video is received makes an entry to db.

Data store:

1. Wide column data store like Cassandra which is scalable for read and write.

Happy Path Data Flow: (Numbering below indicate the actions happening in respective steps in the diagram)



1. Broadcasting machine starts the broadcast using the web application using HLS protocol stream/stream_id, the first request also contains metadata of video like thumbnail, name etc..
2. The request is received by the nearest cluster and load balancer delegates it to video processing Micro service using consistent hashing.
3. The microservice spawns a worker which takes care of encoding the video. The video is compressed and converted into all the expected formats, and is stored in a big cache in chunks. A event is triggered to message broker with the stream id, and other metadata
4. The Video catalogue service understands a new event with help of this event and creates a entry in the data base

Fault Tolerance:

Scenario 1: Lets Says when encoding the video the worker node has crashed

- The system should spawn a new worker node identifying the health status and video encoding is resumed, there will be a little delay in sending out this chunk.

Scenario 2: Lets Say the whole processing service is taken down.

- Since we are using consistent hashing the requests are handled by the next instance.
- Since the service mesh can identify an instance is down and knows which one is broken it can delegate the request to next one with no downtime in video processing.

Scenario 3: Bad internet from broadcaster's end for few mins:

- Unfortunately our broadcaster is low on data speed and due to this there is a overall latency in the video stream to end user.
- In this case, the video processing of higher resolutions will happen eventually and we shall process the low quality videos first and send it out to the client.
- If the above is also taking too long we shall start streaming audio only for couple of secs till the video processing is restored.

Scenario 4: Outrage with cloud provider

- Let's say we are using AWS and there was an outrage in aws systems of bangalore regions.
- The requests are delegated to next nearest cluster(lets assume Chennai). But our video processed so far is bangalore which is now gone.
- For this purpose a regular backup is taken and data is replicated.
- Although a video content of a few seconds is lost during the shift.

Architecture: When video is being requested by one of the clients

Now the video which has been published above has to be served to clients who are requesting them. The above broadcasting activity has happened in the bangalore region of our servers.

The notification service consumes the live stream beginning event from kafka stream and sends out notification to client devices.

Components:

Gateway Load balancer:

1. Responsible for balancing the load across instances.
2. A uuid to be sent in the request by client so we can use a consistent hash function and forward all the requests coming from a single client to a single pod.

Video streaming Micro Service:

1. Responsible for Streaming video to client machines
2. Needs to be **horizontally scalable** in order to meet demand of clients. We never know when a match could get into a super over and gets the attention of the nation.

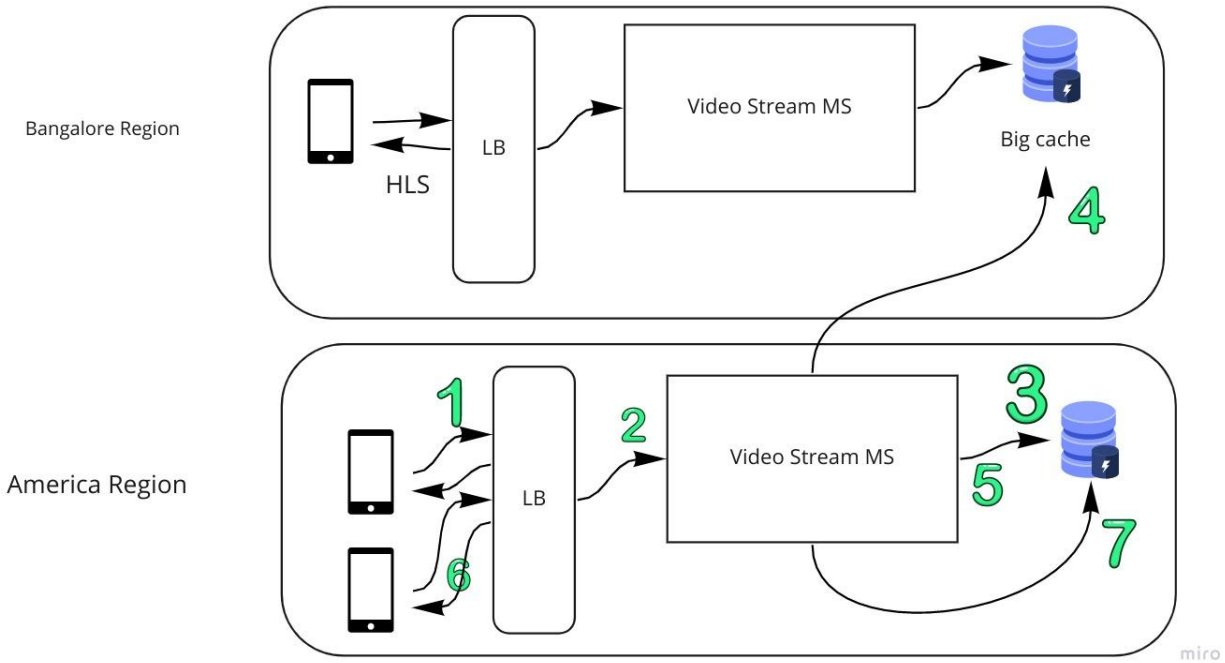
Big cache storage:

Blob store holding the video data

Video catalogue Service:

Processes requests for video metadata

Architecture Diagram



Happy Path Data Flow: (Numbering below indicate the actions happening in respective steps in the diagram)

Scenario 1: A bangalore user has started playing the video.

1. Requests are received by bangalore region video streaming service, looks for chunks in the data store, finds them and sends them out to the user.

Scenario 2: A San fransisco user has started playing the video.

1. Requests are received by North american region GLB
2. GLB delegates the request to Video Streaming service
3. It looks for the video chunk in its corresponding data store and fails to find it.
4. It sends a request looking for the video to bangalore region service,
5. The same is sent to the user and written in data store of North America region,
6. A new user from NA region requests for the chunk
7. This time chunk was found in local region and is sent out to the user

Scenario 3: 100 concurrent San francisco users have started playing the video.

- Let's consider 100 concurrent users from the America region who have started requesting for video. As per above flow, america region doesn't have video in its storage, so it will fetch it from Bangalore region.
- But since 100 of them will fetch at once causing load on the system we will limit it only one fetch from the bangalore system, we will limit refetch for some more secs interval. So bangalore system doesn't experience unwanted load from early america clients

Fault Tolerance:

Scenario 1: Bad internet from user's end

1. System will identify and send out appropriate format of video

Scenario 2: Outrage in San Francisco Region

1. The requests are delegated to next nearest region lets say US East.
2. Since the video streaming service scales horizontally we can keep adding instances to meet the demand of 2 regions.
3. However San Francisco users might face a little latency to fetch the video from the other coast.

Other Microservices:

Identity management service

- Responsible for user logins, user management, issuing session tokens
- Responsible for role based access control, to ensure people with corresponding roles can only perform corresponding actions.

Payment service:

- Responsible for user subscriptions, processing payments

Recommendations engine:

- Spark engine to run jobs for recommendation processing.
- Writes the recommendations to the db.
- Pushes an event for notification service to consume.

Decision Trade offs:

Streaming Protocol RTMP vs SRT vs HLS vs MPEG-DASH

- RTMP has been used for quite a long time, it works over TCP and has little latency.
- SRT on the other hand is new and doesn't support playbacks.
- HLS and MPEG Dash are reliable and have negligible latency. Hence they are best suited for our systems.
- HLS has recently added 4k support and MPEG Dash has compatibility issues with some browsers (source : internet).
- HLS seems to be the option in terms of security and latency.

Video Storage AWS S3 vs Big cache

- Big Cache is open source and can be extended,
- AWS S3 provides a better infrastructure in terms of security and distribution. On other hand Big cache needs to be self maintained.
- To keep it simple for this design i would go with self hosted big cache within our infrastructure and s3 for backups.

Data base selection

- For the above design we only need to store video meta data, if we do not consider user management, payments and recommendations.
- To store video meta data a RDBMS based SQL db would work. But since we need to store user data and map with their streams and keeping the system extendable i would go with **cassandra** which is a highly scable store in terms of reads and writes. Graph based db like neo4j could be a option but since our use case doesn't need complex joins like operation and not many deep relationships I see off, I would not like to have it.

Message Broker Kafka vs Rabbit MQ

- Kafka is scalable and easy to maintain than the later.

API Selection

- API's being exposed to client will could be in Rest
- I would also want to have a gRPC wrapper around the service layer so that service-service calls will be efficient and faster, client platforms with grpc support can also leverage these API's for better performance.

API Design

Video Processor Service:

- HLS stream/stream_id

Notification Service :

- GET /notifications

Video Catalogue Service :

- GET /video - List all videos
- GET /video - List Videos with Query params as filter like name filter for search, date filter etc, and sort options.
- Post /video - Create Video entry in db
- Put /video - Update video record
- Delete /video - Delete video
- Put /video/status - Update video status

Kafka Events :

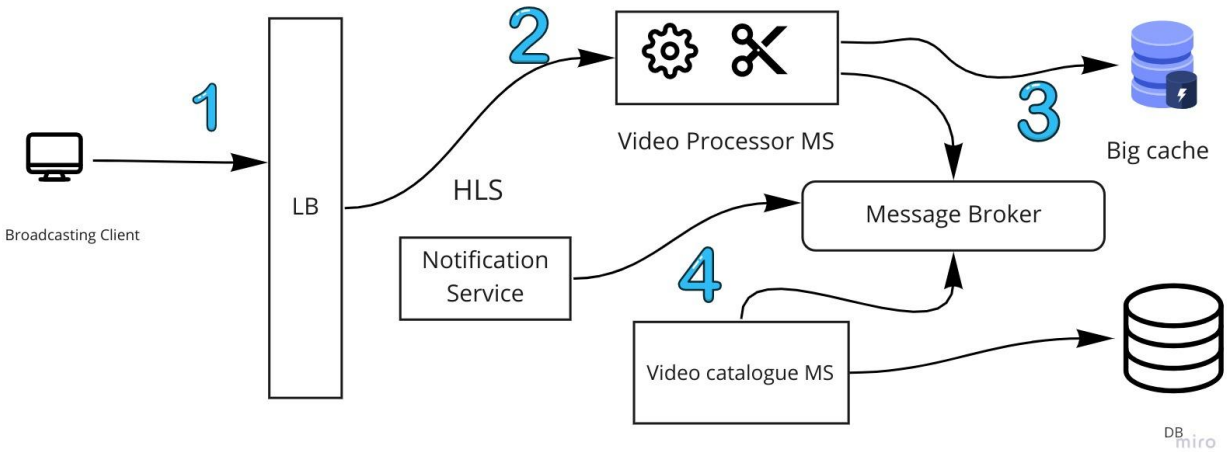
- Video PENDING : Event when stream request is sent to server from broadcast client.
- Video CREATED : Event when Video service writes the entry for Video into db.
- Video STREAM_ACTIVE : Event when video stream started and is ongoing.
- Video STREAM_COMPLETE : Event when video stream has completed
- Video ARCHIVED: Event when video/stream has been archived and unavailable for users.
- Video INACTIVE: Event when video has been permanently taken down from platform

Video Streaming Service:

- HLS Get /stream/stream_id

Extending system to send notifications when Live stream begins:

- Lets add a new micro service for Notifications
- Notification service listens to the events from Message Broker and sends a notification to respective clients.



Extending system to let users add comments/live chat during live stream:

- Comments/chat could go into Video catalogue service
- We could expose an api to add comments to the video.
- We can add a new table in cassandra for comments and map them to stream_id with a timestamp of live stream