highscalability.com

Designing Netflix - High Scalability -

21-26 minutes

This is a guest post by Ankit Sirmorya. Ankit is working as a Machine Learning Lead/Sr. Machine Learning Engineer at Amazon and has led several machine-learning initiatives across the Amazon ecosystem. Ankit has been working on applying machine learning to solve ambiguous business problems and improve customer experience. For instance, he created a platform for experimenting with different hypotheses on Amazon product pages using reinforcement learning techniques. Currently, he is in the Alexa Shopping organization where he is developing machine-learning-based solutions to send personalized reorder hints to *customers for improving their experience.*

Problem Statement

Design a video streaming platform similar to Netflix where content creators can upload their video content and viewers are able to play video on different devices. We should also be able to store user statistics of the videos such as number of views, video watched duration, and so forth.

Gathering Requirements

In Scope

The application should be able to support the following requirements.

- Content Creator should be able to upload content at a designated time.
- The viewers are able to watch the video on different platforms (TV, mobile-app, etc.).
- Users should be able to search videos based on the video titles.
- The system should support sub-titles for the videos as well.

Out of Scope

- The mechanism to recommend personalized videos to different users.
- The billing and subscription model for watching the videos.

Capacity Planning

We need to build an application which should be able to support the traffic at the scale of Netflix. We should also be able to handle the surge of traffic which is often seen when sequel to a popular web-series (e.g. House of Cards, Breaking Bad etc.) comes out. Some of the numbers which are relevant to the capacity planning are listed below.

- Number of active users registered with the application = 100 Million
- Average size of the video content uploaded every minute = 2500 MB
- Total combinations of resolution and codec formats which need to be supported = 10
- Average number of videos users watch daily = 3

Netflix comprises of multiple microservices. However, the playback service responsible for responding to user playback queries will get the maximum traffic. So, it requires the maximum number of servers. We can calculate the number of servers required for handling the playback requests by using the equation mentioned below.

> S*ervers in* playback *microservice*= (#playbacks requested per second$*$ Latency)/ #concurrent connections per server

Let’s assume that the latency (time taken to respond to user requests) of the playback service is 20 milliseconds and each server can support a maximum of 10K connections. Additionally, we need to scale the application for a peak traffic scenario where 75% of active users place playback requests. In such a scenario, we will need a total of 150 servers(=75M*20ms/10K).

> Number of videos watched per second= (#active users * #average videos watched daily)/86400 = 3472(100M * 3/86400)

> Size of the content stored on a daily basis = #average size of video uploaded every min * #pairwise combination of resolutions and codecs * 24* 60 = 36

TB/day (=2500MB * 10 * 24 * 60)

High Level Design

There will primarily be two kinds users of the system: content creators who upload video content, and viewers who watch the video. The entire system can be divided into the following components.

- **Content Distributor Network (CDN)**: It’s responsible for storing the content in the locations which are geographically closest to users. This significantly enhances the user experience as it reduces the network and geographical distances that our video bits must travel during playback. It also reduces the overall demand on upstream network capacity by a great extent.

  FUN FACT: Open Connect is the global customized CDN for Netflix that delivers Netflix TV shows and movies to members world-wide. This essentially is a network of thousands of Open Connect Appliances (OCAs) which store encoded video/image files and is responsible for delivering the playable bits to client devices. OCAs are made up of tuned hardware and software which are deployed on the ISP site and custom tailored to provide optimal customer experience.

- **Control Plane**: This component will be responsible for uploading new content which will eventually be distributed across the CDNs. It will also be responsible for things such as file storage, sharding, data storage and interpretation of relevant telemetry about the playback experience. The major microservices in this component are listed below.
- **CDN Health Checker Service**: This microservice will be responsible to periodically check health of the CDN service, learn about the overall playback experience, and work towards optimizing it.
- **Content Uploader Service**: This microservice will consume the content provided by content generators and distribute it across the CDNs to ensure robustness and optimal playback experience. It will also be responsible for storing the metadata of video content in data-storage.
- **Data Storage**: The video metadata (title, description etc.) is persisted in the data-storage. We will also persist the subtitle information in the optimal database.
- **Data Plane**: This is the component with which the end users interact while playing a video content. This

component gets requests from different media streaming platforms (TV, mobile, tablet etc.) and returns the urls of the CDNs from which the requested files can be served. It will comprise of two major microservices.

- **Playback Service**: This micro-service is responsible for determining the specific files which are required for serving a playback request.
- **Steering Service**: This service determines the optimal CDN urls from which the requested playback can be fetched from.

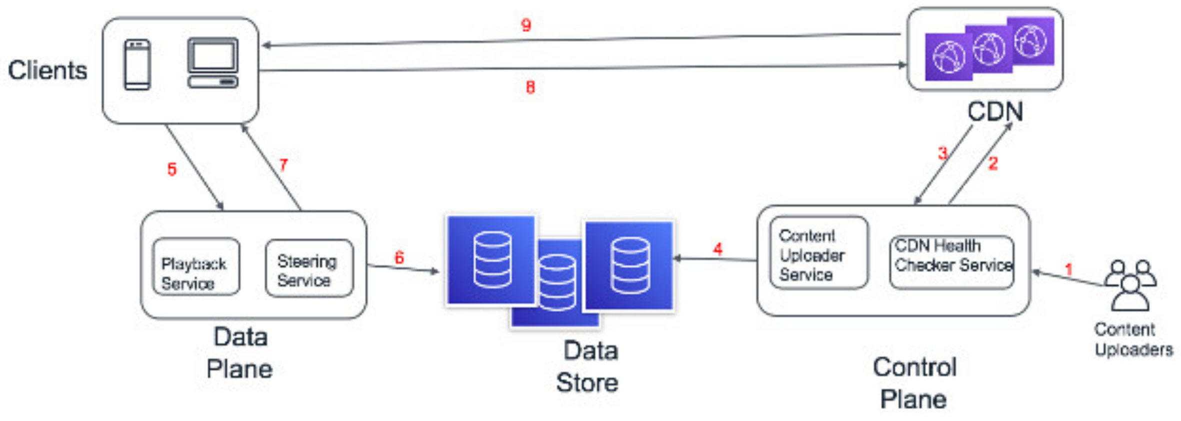


Fig 0: HLD of Netflix

In the image above, we have shown the bird's eye-view of the overall system which should be able to meet all the in-scope requirements. The details of each of the component interactions are listed below.

1. The content creators upload the video content to the control plane.

2. The video content gets uploaded on the CDN which are placed geographically closer to end users.
3. The CDN reports the status to the control plane such as health metrics, what files they have stored, the optimal BGP routes and so forth.
4. The video metadata and the related CDN information gets persisted in the data storage.
5. A user on the client device places the request to play a particular title (TV show or movie).
6. Playback service determines the files which are required to playback a specific title.
7. The Steering service picks the optimal CDNs from which the required files can be fetched from. It generates the urls of such CDNs and provide it to back to the client device.
8. Client device requests the CDNs to serve the requested files.
9. CDN serves the requested files to the client device which gets rendered to the users.

API Design

Video Upload

Path:

POST /video-contents/v1/videos

Body:

```
{
        videoTitle : Title of the video
        videoDescription : Description of the video
        tags : Tags associated with the video
        category : Category of the video, e.g. Movie, TV
Show,
        videoContent: Stream of video content to be
uploaded
}
```

Search Video

Path:

GET /video-contents/v1/search-query/

Query Param:

```
{
        user-location: location of the user performing
```

search
}

Stream Video

Path:

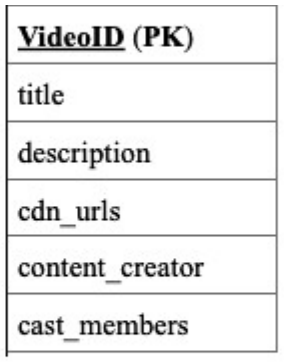
GET /video-contents/v1/videos/

Query Param:

```
{
        offset: Time in seconds from the beginning of the
video
}
```

Data Model

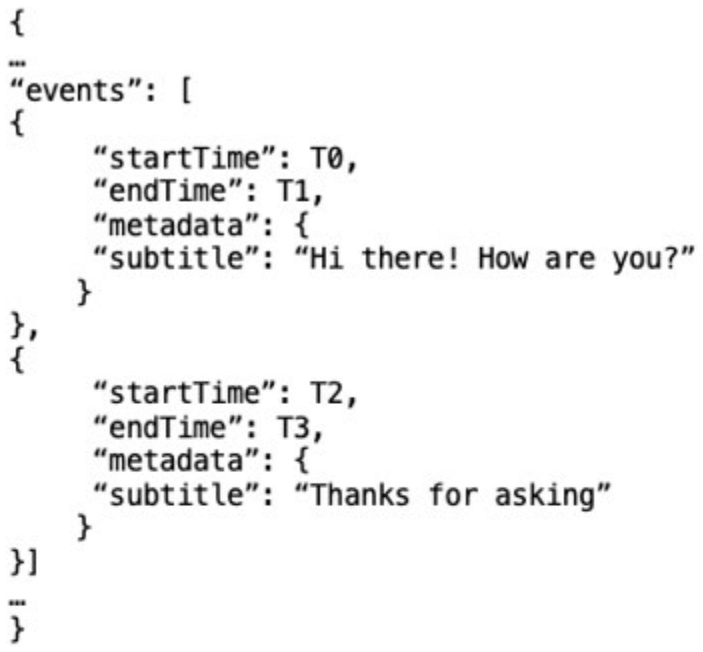
Within the scope of this problem, we need to persist video metadata and its subtitles in the database. The video metadata can be stored in an aggregate-oriented database and given that the values within the aggregate may be updated frequently, we can use a document-based store like MongoDB to store this information. The data-model for storing the metadata is shown in the table below.



VideoID (PK)
title
description
cdn_urls
content_creator
cast_members

Table: Video Metadata Data Model

We can use a time-series database such as OpenTSDB, which builds on top of Cassandra, to store the sub-titles. We have shown below a snippet of the data-model which can be used to store video sub-titles. In this model(let’s call it Media Document), we have provided an event-based representation where each event occupies a time-interval on the timeline.



```
{
…
“events”: [
{
      “startTime”: T0,
      “endTime”: T1,
      “metadata”: {
      “subtitle”: “Hi there! How are you?”
     }
},
{
      “startTime”: T2,
      “endTime”: T3,
      “metadata”: {
      “subtitle”: “Thanks for asking”
     }
}]
…
}
```

FUN FACT: In this talk, Rohit Puri from Netflix, talks about the Netflix Media Database(NMDB) which is modeled around the notion of a media timeline with spatial properties. NMDB is desired to be a highly-scalable, multi-tenant, media metadata system which can serve near real-time queries and can serve high read/write throughput. The structure of the media timeline data model is called a “**Media Document**”.

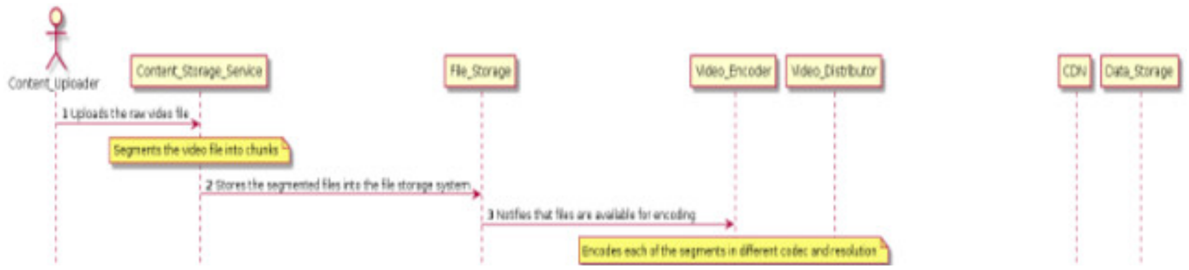
Component Design

Control Plane

This component will mainly comprise of three modules: Content Uploader, CDN Health Checker, and Title Indexer. Each of these modules will be a micro-service performing a specific task. We have covered details of each of these modules in the section below.

Content Uploader

This module is executed when a content creator uploads a content. It is responsible for distributing the content on CDN to provide optimal customer experience.


Content_Uploader
Content_Storage_Service
File_Storage
Video_Encoder
Video_Distributor
CDN
Data_Storage
1 Uploads the raw video file
Segments the video file into chunks
2 Stores the segmented files into the file storage system
3 Notifies that files are available for encoding
Encodes each of the segments in different codec and resolution

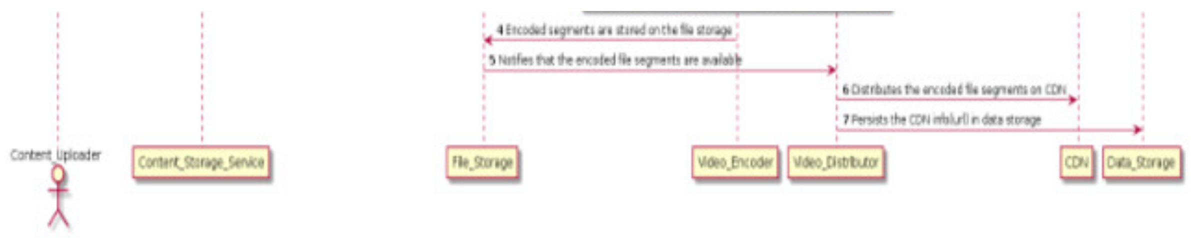

4 Encoded segments are stored on the file storage
5 Notifies that the encoded file segments are available
6 Distributes the encoded file segments on CDN
7 Persists the CDN info(url) in data storage
Content Uploader
Content_Storage_Service
File_Storage
Video_Encoder
Video_Distributor
CDN
Data_Storage

Fig 1: Sequence Diagram of Content Upload Operation

The diagram above depicts the sequence of operations which gets executed when content creators upload the video content (TV Show or movie).

1. The content creator uploads the raw video content which can be TV Show or movie.
2. The Content_Storage_Service segments the raw video file into chunks and persists those segments on the file storage system.
3. The Video_Encoder encodes each of the segments in different codec and resolution.
4. The encoded file segments are stored in the file storage.
5. The Video_Distributor reads the encoded file segments from the distributed file storage system.
6. The Video_Distributor distributes the encoded file segments in CDN.
7. The Video_Distributor persists the CDN url links of the videos in the data_storage.

Video Encoder

The encoder works by splitting the video file into smaller video segments. These video segments are encoded in all possible combinations of codecs and resolutions. In our example, we can plan on supporting four codecs(Cinepak, MPEG-2, H.264, VP8) and three different resolutions(240p, 480p, 720p). This implies that each video segment gets encoded in a total of 12 formats(4 codecs * 3 resolutions). These encoded video segments are distributed across the CDN and the CDN url is maintained in the data store. The playback api is responsible for finding the most optimal CDN url based on the input parameters(client's device, bandwidth, and so forth) of user requests.

FUN FACT: Netflix's media processing platform is used for video encoding(FFmpeg), title image generation, media processing(Archer) and so forth. They have developed a tool called MezzFS which collects metrics on data throughput, download efficiency, resource usage, etc. in Atlas, Netflix's in-memory time-series database. They use this data for developing optimizations such as replays and adaptive buffering.

CDN Health Checker

This module ingests the health metrics of the CDNs and persists them in the data storage. This data is used by the data plane to get optimal CDN urls when users request playback.

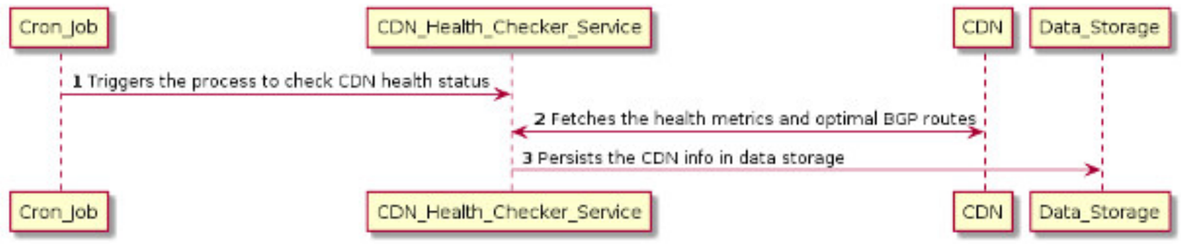

Cron_Job
CDN_Health_Checker_Service
CDN
Data_Storage
1 Triggers the process to check CDN health status
2 Fetches the health metrics and optimal BGP routes
3 Persists the CDN info in data storage
Cron_Job
CDN_Health_Checker_Service
CDN
Data_Storage

Fig 2: Sequence Diagram to check CDN Health Metrics

In the image above, we have shown the sequence of operations which gets executed to get statistics around the CDN health metrics and the BGP routes. The details about each of the steps in the sequence diagram are listed below.

1. The cron job triggers the microservice (CDN_Health_Checker_Service) responsible for checking the health of CDNs.
2. The CDN_Health_Checker_Service is responsible for checking the health of CDNs and collect health metrics and other information.
3. The CDN_Health_Checker_Service persists the CDN info in the data store which is then used in the data-plane to find the optimal CDNs from which files can be served based on their file availability, health, and

network proximity to the client.

Title Indexer

This module is responsible for creating the indexes of the video titles and updates them in the elastic search to enable faster content discovery for end users.

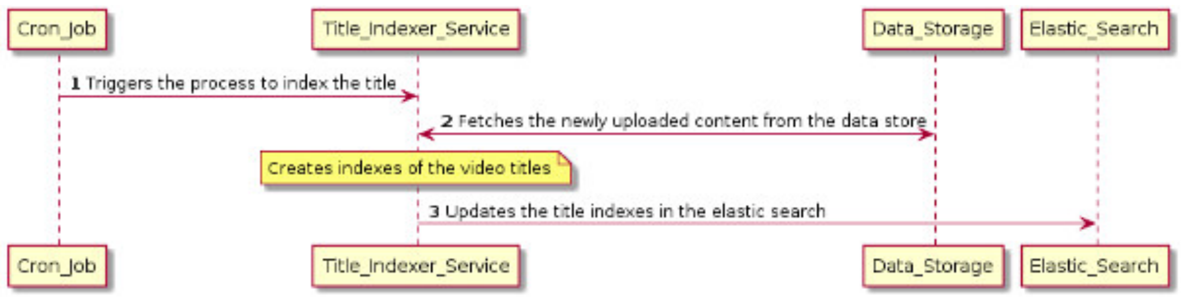

Cron_Job
Title_Indexer_Service
Data_Storage
Elastic_Search
1 Triggers the process to index the title
2 Fetches the newly uploaded content from the data store
Creates indexes of the video titles
3 Updates the title indexes in the elastic search
Cron_Job
Title_Indexer_Service
Data_Storage
Elastic_Search

Fig 3: Sequence Diagram to store indexed titles on ElasticSearch

The details of the sequence of operations required for indexing the video titles for searching video content are listed below.

1. The cron-job triggers the Title_Indexer_Service to index the video titles.
2. The Title_Indexer_Service fetches the newly uploaded content from the data-store and applies the business rules to create indexes for the video titles.
3. Title_Indexer_Service updates Elastic_Search with the indexes of the video titles making the titles easily searchable.

Data Plane

This component will be processing the user requests in real-time and will comprise of two major workflows: Playback Workflow and Content Discovery Workflow.

Playback Workflow

This workflow is responsible for orchestrating operations when a user places a playback request. It co-ordinates between different microservices such as Authorization Service (for checking user authorization and licensing), Steering Service (for deciding the best playback experience) and Playback Experience Service (for tracking the events to measure playback experience). Steering Service ensures the best customer experience by finding the most optimal CDN url based on user request such as user's device, bandwidth and so forth. The orchestration process will be handled by the Playback_Service as shown in the image below.

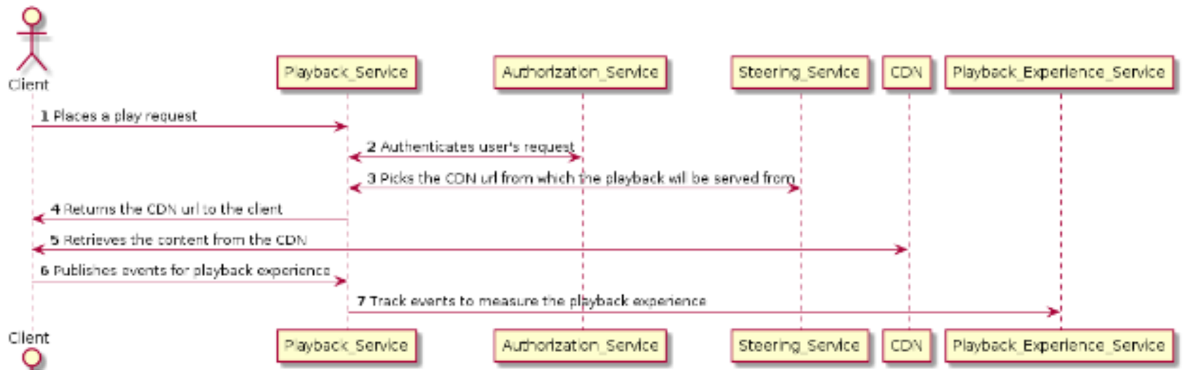

Client
Playback_Service
Authorization_Service
Steering_Service
CDN
Playback_Experience_Service
1 Places a play request
2 Authenticates user's request
3 Picks the CDN url from which the playback wil be served from
4 Returns the CDN url to the client
5 Retrieves the content from the CDN
6 Publishes events for playback experience
7 Track events to measure the playback experience
Client
Playback_Service
Authorization_Service
Steering_Service
CDN
Playback_Experience_Service

Fig 4: Sequence Diagram of Playback Service

The details about each of the steps in the sequence diagram is listed below.

1. The client places a request to playback a video which gets directed to the Playback_Service.
2. The Playback_Service calls the Authorization_Service to authenticate users request.
3. The Playback_Service calls the Steering_Service to pick the CDN url from which the playback can be served.
4. The CDN url is returned to the client(mobile/TV).
5. The client retrieves the content from CDN.
6. The client publishes the events for the playback experience to the Playback_Service.
7. The Playback_Service tracks events to measure the playback experience by calling the Playback_Experience_Service.

FUN FACT: As mentioned in this talk by Netflix Engineer Suudhan Rangarajan, gRPC is used as the framework for communication between different micro-services at Netflix. The advantages it provides over

REST include: bi-directional streaming, minimum operational coupling and support across languages and platforms.

Content Lookup Workflow

This workflow is triggered when user searches for a video title and comprises of two microservices: Content Discovery Service and Content Similarity Service. The Content Discovery Service gets invoked when user requests for the video title. On the other hand, the Content Similarity Service returns the list of similar video title if the exact video title doesn't exist in our data-store.

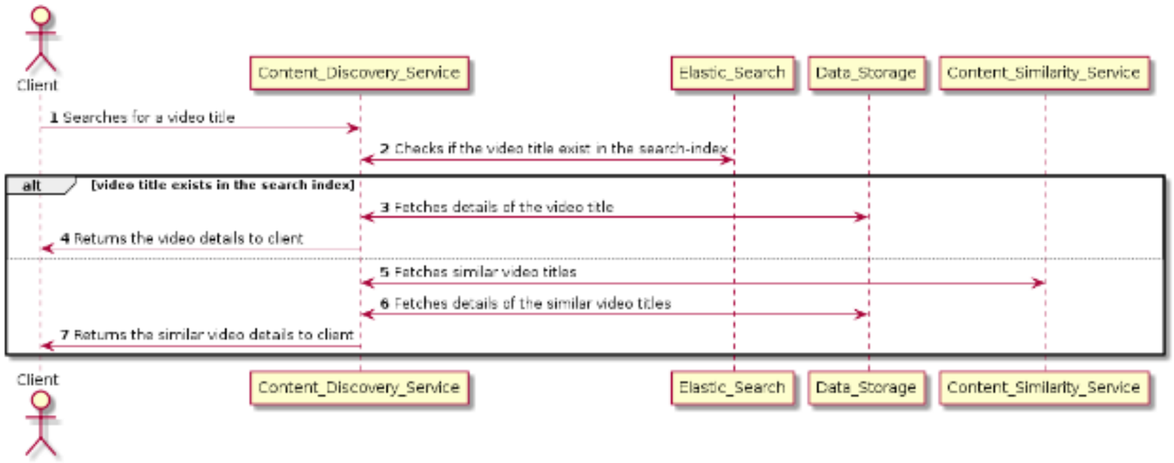

Client
Content_Discovery_Service
Elastic_Search
Data_Storage
Content_Similarity_Service
1 Searches for a video title
2 Checks if the video title exist in the search-index
alt
[video title exists in the search index]
3 Fetches details of the video title
4 Returns the video details to client
5 Fetches similar video titles
6 Fetches details of the similar video titles
7 Returns the similar video details to client

Fig 5: Sequence Diagram of Content Lookup Workflow

We have listed below details of each of the steps involved in the Content Lookup Workflow.

1. The client searches for a video title.

2. The Content Discovery Service (CDS) queries the Elastic Search to check if the video title exists in our database.
3. If the video title can be found in the elastic search then CDS fetches the details of the video from the data store.
4. The video details are returned to the client.
5. CDS queries the Content Similarity Service (CSS) if the title doesn’t exist in our database. CSS returns the list of similar video titles to CDS.
6. CDS fetches the video details from the data-store for those similar video titles.
7. CDS returns the similar video details to the client.

Optimizations

We can work towards optimizing the latency of the Playback Workflow by caching the CDN information. This cache will be used by the steering service to pick the CDNs from which the video content would be served. We can further enhance performance of the playback operation by making the architecture asynchronous. Let’s try to understand it further using example of the playback api(getPlayData()) which requires customer(getCustomerInfo()) and device information(getDeviceInfo()) to process(decidePlayData()) a video playback request. Suppose each of the three operations(getCustomerInfo(), getDeviceInfo(), and decidePlayData()) depend on different microservices.

The synchronous implementation of the getPlayData() operation will look similar to the code snippet shown below. Such an architecture will comprise of two types of thread-pools: request-handler thread-pool and client thread-pools (for each of the micro-services). For each playback request, an execution thread from the request-response thread-pool gets blocked until the getPlayData() call gets finished. Each time getPlayData() is invoked, an execution thread(from the request-handler thread-pool) interacts with the client thread-pools of the dependent microservices. It is blocked until the execution is completely finished. It works for a simple request/response model where latency isn't a concern, and the number of clients is limited.

```
PlayData getPlayData(String customerId, String titleId,
String deviceId) {
        CustomerInfo custInfo =
getCustomerInfo(customerId);
        DeviceInfo deviceInfo =
getDeviceInfo(deviceId);
        PlayData playData = decidePlayData(custInfo,
deviceInfo, titleId);
        return playData;
}
```

One way to scale out the playback operation is to split the operation into independent processes which can be executed in-parallel and re-assembled together. This can be done by using an asynchronous architecture comprising of event-loops for handling request-responses and client-interactions along-with worker threads. We have shown the asynchronous processing of the playback requests in the image below.

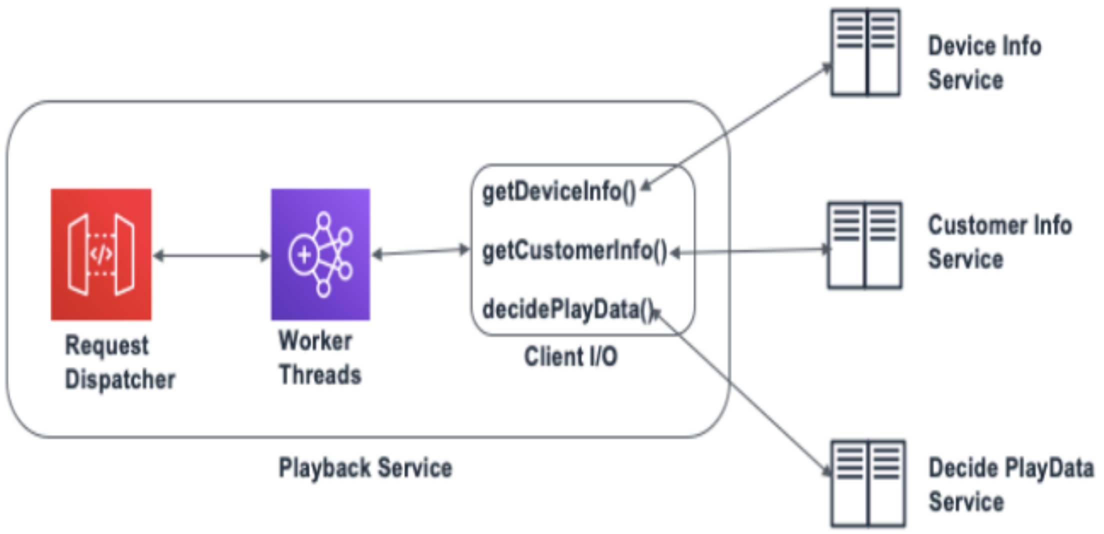


Fig 6: Asynchronous architecture of Playback API

We have shown below the tweaked the code-snippet to effectively leverage the asynchronous architecture. For every playback request, the request-handler event pool triggers a worker thread to set up the entire execution flow. After that, one of the worker threads fetches customer-info from the related micro-service and another thread fetches the device information. Once a response is returned by both the worker threads, a separate execution unit bundles the two responses together and uses them for the decidePlayData() call. In such a process, all the context is passed as messages between separate threads. The asynchronous architecture not only helps by effectively leveraging the available computational resources but also reducing the latency.

```
PlayData getPlayData(String customerId, String titleId,
String deviceId) {
       Zip(getCustomerInfo(customerId),
             getDeviceInfo(deviceId),
              (custInfo, deviceInfo) ->
decidePlayData(custInfo, deviceInfo, titleId)
               );
}
```

Addressing Bottlenecks

The usage of micro-services comes with the caveat of efficiently handling fallbacks, retries and time-outs while calling other services. We can address the bottlenecks of using distributed systems by using the concepts of Chaos Engineering, interestingly devised at Netflix. We can use tools such as Chaos Monkey which randomly terminates instances in production to ensure that services are resilient to instance failures.

We may introduce chaos in the system by using the concepts of Failure Injection Testing(FIT). This can be done by either introducing latency in the I/O calls or by injecting faults while calling other services. After that, we can implement fallback strategies by either returning latest cached data from the failing service or using a fallback microservice. We can also use libraries such as Hystrix for isolating the points of access between failing services. Hystrix acts as circuit breakers if the error threshold gets breached. We should also ensure that the retry time-outs, service call time-outs, and the hystrix time-outs are in sync.

FUN FACT: In this presentation by Nora Jones (Chaos Engineer at Netflix), the importance and different strategies of resilience testing at Netflix is discussed at length. She has provided key pointers which engineers should keep in mind while designing microservices for resiliency and to ensure that optimal design decisions are in place on a continuous basis.

Extended Requirements

A common issue observed while streaming a video is that the subtitle appears on top of a text in the video (called the text-on-text issue). The issue is illustrated in the image below. How can we extend the current solution and the data model to detect this issue?

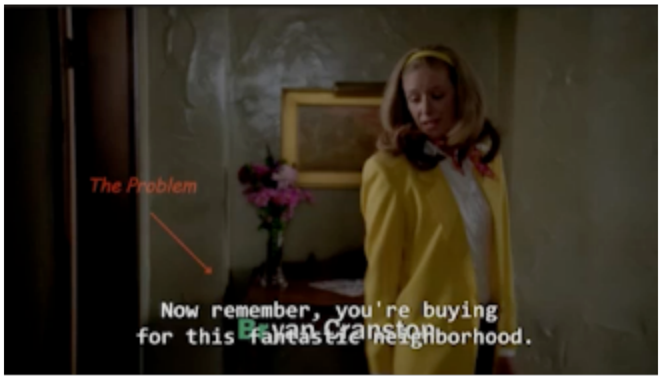


Fig 7: Example of Text-on-Text Issue

We can extend the existing Media Document solution (used for video subtitles) to store the video media information as well. We can then run Text-in-video detection and subtitle positioning algorithms on the media document data store and persist the results as separate indexes. After that, these indexes will be queried by the Text-on-Text detection application to identify any overlap, which will detect the text-on-text issue.

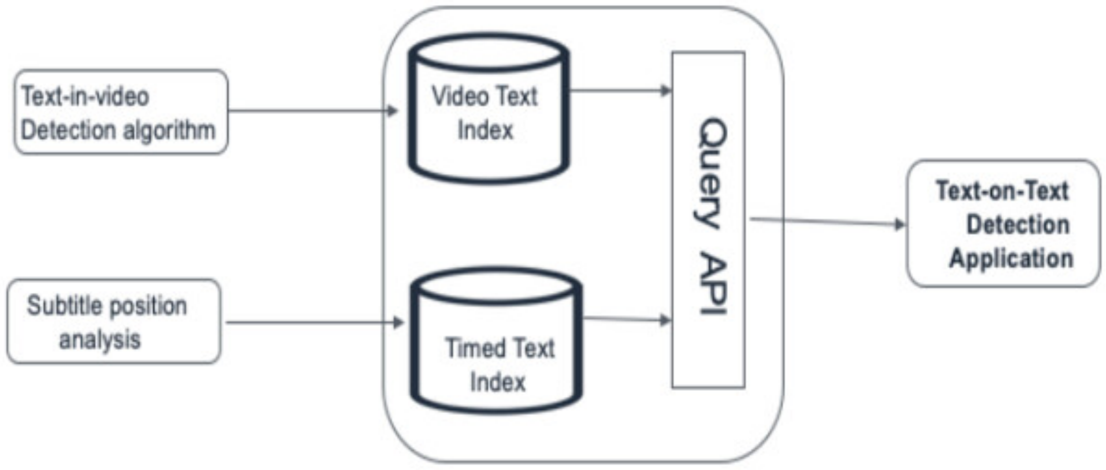

Text-in-video Detection algorithm
Subtitle position analysis
Video Text Index
Timed Text Index
Query API
Text-on-Text Detection Application

Fig8 : Text-on-Text Detection Application Flow

References

General

[Video of a Talk on this Paper]

Netflix originals

https://www.youtube.com/watch?v=CZ3wIuvmHeM

https://www.youtube.com/watch?v=RWyZkNzvC-c

https://www.youtube.com/watch?v=LkLLpYdDINA

https://www.youtube.com/watch?v=6oPj-DW09DU

https://www.youtube.com/channel/UC8MdvSjinN761VUtnUjHwJw

Architecture

https://openconnect.netflix.com/Open-Connect-Overview.pdf

https://medium.com/netflix-techblog/mezzfs-mounting-object-storage-in-netflixs-media-processing-platform-cda01c446ba

https://medium.com/netflix-techblog/simplifying-media-innovation-at-netflix-with-archer-3f8cbb0e2bcb

https://www.youtube.com/watch?v=CZ3wIuvmHeM

https://www.youtube.com/watch?v=6oPj-DW09DU

Data Model

https://www.youtube.com/watch?v=OQK3E21BEn8

https://medium.com/netflix-techblog/implementing-the-netflix-media-database-53b5a840b42a

Fun Facts

Chaos Engineering:

https://www.youtube.com/watch?v=RWyZkNzvC-c

Other

https://www.youtube.com/watch?v=psQzyFfsUGU

https://www.youtube.com/watch?v=x9Hrn0oNmJM

Written

https://medium.com/@narengowda/netflix-system-design-dbec30fede8d

https://openconnect.netflix.com/Open-Connect-Overview.pdf

https://medium.com/netflix-techblog/implementing-the-netflix-media-database-53b5a840b42a