24. Netflix

Let's design a Netflix like video streaming service, similar to services like Amazon Prime Video, Disney Plus, Hulu, Youtube, Vimeo, etc.

What is Netflix?

Netflix is a subscription-based streaming service that allows its members to watch TV shows and movies on an internet-connected device. It is available on platforms such as the Web, iOS, Android, TV, etc.

Requirements

Our system should meet the following requirements:

Functional requirements

- Users should be able to stream and share videos.
- The content team (or users in YouTube's case) should be able to upload new videos (movies, tv shows episodes, and other content).
- Users should be able to search for videos using titles or tags.
- Users should be able to comment on a video similar to YouTube.
- Non-Functional requirements
- High availability with minimal latency.
- High reliability, no uploads should be lost.
- The system should be scalable and efficient.
- Extended requirements
- Certain content should be geo-blocked.
- Resume video playback from the point user left off.
- Record metrics and analytics of videos.
- Estimation and Constraints

Let's start with the estimation and constraints.

Note: Make sure to check any scale or traffic-related assumptions with your interviewer.

Traffic

This will be a read-heavy system, let us assume we have 1 billion total users with 200 million daily active users (DAU), and on average each user watches 5 videos a day. This gives us 1 billion videos watched per day.

200 Million × 5 videos=1 billion/day

Assuming a 200:1 read/write ratio, about 5 million videos will be uploaded every day.

200 ×1 billion=5 million/day

What would be Requests Per Second (RPS) for our system?

1 billion requests per day translate into 12K requests per second.

1 billion(24 hrs×3600 seconds)=~ 12K requests/second

Storage

If we assume each video is 100 MB on average, we will require about 500 TB of storage every day.

5 million× 100 MB=500 TB/day

And for 10 years, we will require an astounding 1,825 PB of storage.

500 TB×365 days X 10 years=1,825 PB

Bandwidth

As our system is handling 500 TB of ingress every day, we will require a minimum bandwidth of around 5.8 GB per second.

500 TB(24 hrs × 3600 seconds)= 5.8 GB/second

High-level estimate

Type	Estimate
Daily active users (DAU)	200 million
Requests per second (RPS)	12K/s
Storage (per day)	~500 TB
Storage (10 years)	~1,825 PB
Bandwidth	~5.8 GB/s

Data model design

This is the general data model which reflects our requirements.

netflix-datamodel

We have the following tables:

- **Users**: This table will contain a user's information such as name, email, dob, and other details.
- **Videos**: As the name suggests, this table will store videos and their properties such as title, streamURL, tags, etc. We will also store the corresponding userID.
- **Tags**: This table will simply store tags associated with a video.
- **Views**: This table helps us to store all the views received on a video.

- **Comments** : This table stores all the comments received on a video (like YouTube).

What kind of database should we use?

While our data model seems quite relational, we don't necessarily need to store everything in a single database, as this can limit our scalability and quickly become a bottleneck.

We will split the data between different services each having ownership over a particular table. Then we can use a relational database such as PostgreSQL or a distributed NoSQL database such as Apache Cassandra for our use case.

API design

Let us do a basic API design for our services:

Upload a video

Given a byte stream, this API enables video to be uploaded to our service.

uploadVideo(title: string, description: string, data: Stream<byte>, tags?: string[]): boolean

Parameters

- Title (string): Title of the new video.
- Description (string): Description of the new video.
- Data (byte[]): Byte stream of the video data.
- Tags (string[]): Tags for the video (optional).
- Returns
- Result (boolean): Represents whether the operation was successful or not.

Streaming a video

This API allows our users to stream a video with the preferred codec and resolution.

streamVideo(videoID: UUID, codec: Enum<string>, resolution: Tuple<int>, offset?: int): VideoStream

Parameters

- Video ID (UUID): ID of the video that needs to be streamed.
- Codec (Enum<string>): Required codec of the requested video, such as h.265, h.264, VP9, etc.
- Resolution (Tuple<int>): Resolution of the requested video.
- Offset (int): Offset of the video stream in seconds to stream data from any point in the video (optional).
- Returns
- Stream (VideoStream): Data stream of the requested video.

Search for a video

This API will enable our users to search for a video based on its title or tags.

searchVideo(query: string, nextPage?: string): Video[]

Parameters

- Query (string): Search query from the user.
- Next Page (string): Token for the next page, this can be used for pagination (optional).
- Returns
- Videos (Video[]): All the videos available for a particular search query.

Add a comment

This API will allow our users to post a comment on a video (like YouTube).

comment(videoID: UUID, comment: string): boolean

Parameters

- VideoID (UUID): ID of the video user wants to comment on.
- Comment (string): The text content of the comment.
- Returns
- Result (boolean): Represents whether the operation was successful or not.

High-level design

Now let us do a high-level design of our system.

Architecture

We will be using microservices architecture since it will make it easier to horizontally scale and decouple our services. Each service will have ownership of its own data model. Let's try to divide our system into some core services.

User Service

This service handles user-related concerns such as authentication and user information.

Stream Service

The stream service will handle video streaming-related functionality.

Search Service

The service is responsible for handling search-related functionality. It will be discussed in detail separately.

Media service

This service will handle the video uploads and processing. It will be discussed in detail separately.

Analytics Service

This service will be used for metrics and analytics use cases.

What about inter-service communication and service discovery?

Since our architecture is microservices-based, services will be communicating with each other as well. Generally, REST or HTTP performs well but we can further improve the performance using gRPC which is more lightweight and efficient.

Service discovery is another thing we will have to take into account. We can also use a service mesh that enables managed, observable, and secure communication between individual services.

Note: Learn more about REST, GraphQL, gRPC and how they compare with each other.

Video processing

video-processing-pipeline

There are so many variables in play when it comes to processing a video. For example, an average data size of two-hour raw 8K footage from a high-end camera can easily be up to 4 TB, thus we need to have some kind of processing to reduce both storage and delivery costs.

Here's how we can process videos once they're uploaded by the content team (or users in YouTube's case) and are queued for processing in our message queue.

Let's discuss how this works:

File Chunker

This is the first step of our processing pipeline. File chunking is the process of splitting a file into smaller pieces called chunks. It can help us eliminate duplicate copies of repeating data on storage, and reduces the amount of data sent over the network by only selecting changed chunks.

Usually, a video file can be split into equal size chunks based on timestamps but Netflix instead splits chunks based on scenes. This slight variation becomes a huge factor for a better user experience since whenever the client requests a chunk from the server, there is a lower chance of interruption as a complete scene will be retrieved.

Content Filter

This step checks if the video adheres to the content policy of the platform. This can be pre-approved as in the case of Netflix according to content rating of the media or can be strictly enforced like by YouTube.

This entire process is done by a machine learning model which performs copyright, piracy, and NSFW checks. If issues are found, we can push the task to a dead-letter queue (DLQ) and someone from the moderation team can do further inspection.

Transcoder

Transcoding is a process in which the original data is decoded to an intermediate uncompressed format, which is then encoded into the target format. This process uses different codecs to perform bitrate adjustment, image downsampling, or re-encoding the media.

This results in a smaller size file and a much more optimized format for the target devices. Standalone solutions such as FFmpeg or cloud-based solutions like AWS Elemental MediaConvert can be used to implement this step of the pipeline.

Quality Conversion

This is the last step of the processing pipeline and as the name suggests, this step handles the conversion of the transcoded media from the previous step into different resolutions such as 4K, 1440p, 1080p, 720p, etc.

It allows us to fetch the desired quality of the video as per the user's request, and once the media file finishes processing, it gets uploaded to a distributed file storage such as HDFS, GlusterFS, or an object storage such as Amazon S3 for later retrieval during streaming.

Note: We can add additional steps such as subtitles and thumbnails generation as part of our pipeline.

Why are we using a message queue?

Processing videos as a long-running task and using a message queue makes much more sense. It also decouples our video processing pipeline from the upload functionality. We can use something like Amazon SQS or RabbitMQ to support this.

Video streaming

Video streaming is a challenging task from both the client and server perspectives. Moreover, internet connection speeds vary quite a lot between different users. To make sure users don't re-fetch the same content, we can use a Content Delivery Network (CDN).

Netflix takes this a step further with its Open Connect program. In this approach, they partner with thousands of Internet Service Providers (ISPs) to localize their traffic and deliver their content more efficiently.

What is the difference between Netflix's Open Connect and a traditional Content Delivery Network (CDN)?

Netflix Open Connect is a purpose-built Content Delivery Network (CDN) responsible for serving Netflix's video traffic. Around 95% of the traffic globally is delivered via direct connections between Open Connect and the ISPs their customers use to access the internet.

Currently, they have Open Connect Appliances (OCAs) in over 1000 separate locations around the world. In case of issues, Open Connect Appliances (OCAs) can failover, and the traffic can be re-routed to Netflix servers.

Additionally, we can use Adaptive bitrate streaming protocols such as HTTP Live Streaming (HLS) which is designed for reliability and it dynamically adapts to network conditions by optimizing playback for the available speed of the connections.

Lastly, for playing the video from where the user left off (part of our extended requirements), we can simply use the offset property we stored in the views table to retrieve the scene chunk at that particular timestamp and resume the playback for the user.

Searching

Sometimes traditional DBMS are not performant enough, we need something which allows us to store, search, and analyze huge volumes of data quickly and in near real-time and give results within milliseconds. Elasticsearch can help us with this use case.

Elasticsearch is a distributed, free and open search and analytics engine for all types of data, including textual, numerical, geospatial, structured, and unstructured. It is built on top of Apache Lucene.

How do we identify trending content?

Trending functionality will be based on top of the search functionality. We can cache the most frequently searched queries in the last N seconds and update them every M seconds using some sort of batch job mechanism.

Sharing

Sharing content is an important part of any platform, for this, we can have some sort of URL shortener service in place that can generate short URLs for the users to share.

For more details, refer to the URL Shortener system design.

Detailed design

It's time to discuss our design decisions in detail.

Data Partitioning

To scale out our databases we will need to partition our data. Horizontal partitioning (aka Sharding) can be a good first step. We can use partitions schemes such as:

- Hash-Based Partitioning
- List-Based Partitioning
- Range Based Partitioning
- Composite Partitioning

The above approaches can still cause uneven data and load distribution, we can solve this using Consistent hashing.

For more details, refer to Sharding and Consistent Hashing.

Geo-blocking

Platforms like Netflix and YouTube use Geo-blocking to restrict content in certain geographical areas or countries. This is primarily done due to legal distribution laws that Netflix has to adhere to when they make a deal with the production and distribution companies. In the case of YouTube, this will be controlled by the user during the publishing of the content.

We can determine the user's location either using their IP or region settings in their profile then use services like Amazon CloudFront which supports a geographic restrictions feature or a geolocation routing policy with Amazon Route53 to restrict the content and re-route the user to an error page if the content is not available in that particular region or country.

Recommendations

Netflix uses a machine learning model which uses the user's viewing history to predict what the user might like to watch next, an algorithm like Collaborative Filtering can be used.

However, Netflix (like YouTube) uses its own algorithm called Netflix Recommendation Engine which can track several data points such as:

- User profile information like age, gender, and location.
- Browsing and scrolling behavior of the user.
- Time and date a user watched a title.
- The device which was used to stream the content.
- The number of searches and what terms were searched.
- For more detail, refer to Netflix recommendation research.

Metrics and Analytics

Recording analytics and metrics is one of our extended requirements. We can capture the data from different services and run analytics on the data using Apache Spark which is an open-source unified analytics engine for large-scale data processing. Additionally, we can store critical metadata in the views table to increase data points within our data.

Caching

In a streaming platform, caching is important. We have to be able to cache as much static media content as possible to improve user experience. We can use solutions like Redis or Memcached but what kind of cache eviction policy would best fit our needs?

Which cache eviction policy to use?

Least Recently Used (LRU) can be a good policy for our system. In this policy, we discard the least recently used key first.

How to handle cache miss?

Whenever there is a cache miss, our servers can hit the database directly and update the cache with the new entries.

For more details, refer to Caching.

Media streaming and storage

As most of our storage space will be used for storing media files such as thumbnails and videos. Per our discussion earlier, the media service will be handling both the upload and processing of media files.

We will use distributed file storage such as HDFS, GlusterFS, or an object storage such as Amazon S3 for storage and streaming of the content.

Content Delivery Network (CDN)

Content Delivery Network (CDN) increases content availability and redundancy while reducing bandwidth costs. Generally, static files such as images, and videos are served from CDN. We can use services like Amazon CloudFront or Cloudflare CDN for this use case.

Identify and resolve bottlenecks

netflix-advanced-design

Let us identify and resolve bottlenecks such as single points of failure in our design:

- "What if one of our services crashes?"
- "How will we distribute our traffic between our components?"
- "How can we reduce the load on our database?"
- "How to improve the availability of our cache?"

To make our system more resilient we can do the following:

- Running multiple instances of each of our services.
- Introducing load balancers between clients, servers, databases, and cache servers.
- Using multiple read replicas for our databases.
- Multiple instances and replicas for our distributed cache.

Assumptions:

We are using a microservices architecture, so each service will be encapsulated in its own component.

We will use a relational database (e.g., PostgreSQL) for most parts of the system (users, videos, comments), and NoSQL (e.g., Cassandra or Elasticsearch) for search functionalities.

1. User Service - Handling User Authentication and Profile

Model for User:

```
public class User {

  @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;


    private String username;

    private String email;

    private String password;

    private String dob;


    @OneToMany(mappedBy = "user")

    private List<Video> videos;


    // Getters and Setters

}
```

User Repository (Using Spring Data JPA for Persistence):

```
public interface UserRepository extends JpaRepository<User, Long> {

    Optional<User> findByEmail(String email);

    Optional<User> findByUsername(String username);

}
```

User Service:

```
@Service

public class UserService {


    @Autowired
    private UserRepository userRepository;


    public User registerUser(User user) {
        return userRepository.save(user);
    }


    public Optional<User> getUserByEmail(String email) {
        return userRepository.findByEmail(email);
    }


    public Optional<User> getUserByUsername(String username) {
        return userRepository.findByUsername(username);
    }
}
```

2. Video Service - Handling Video Uploads and Metadata

Model for Video:

```
@Entity
public class Video {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;


    private String title;
    private String description;
    private String streamUrl;


    @ManyToOne

    @JoinColumn(name = "user_id")

    private User user;


    @ManyToMany

    @JoinTable(

        name = "video_tags",

        joinColumns = @JoinColumn(name = "video_id"),

        inverseJoinColumns = @JoinColumn(name = "tag_id")

    )

    private List<Tag> tags;


    // Getters and Setters

}
```

Video Repository:

```
public interface VideoRepository extends JpaRepository<Video, Long> {

    List<Video> findByTitleContainingOrTagsNameContaining(String title, String tagName);

}
```

Video Service:

```
@Service
public class VideoService {

    @Autowired
    private VideoRepository videoRepository;

    @Autowired
    private S3Service s3Service;  // For storing videos in AWS S3

    public Video uploadVideo(Video video, byte[] videoData) {
        // Store the video in S3 and get the URL
        String videoUrl = s3Service.uploadVideo(videoData);
        video.setStreamUrl(videoUrl);

        return videoRepository.save(video);
    }

    public List<Video> searchVideos(String query) {
        return videoRepository.findByTitleContainingOrTagsNameContaining(query, query);
    }
}
```

3. Video Streaming Service - Handling Video Streaming Requests

Video Stream API:

```
@RestController

@RequestMapping("/api/videos")

public class VideoController {


    @Autowired

    private VideoService videoService;


    @GetMapping("/stream/{videoId}")

    public ResponseEntity<Resource> streamVideo(@PathVariable Long videoId,

                                                @RequestParam String codec,

                                                @RequestParam String resolution,

                                                @RequestParam(defaultValue = "0") int offset) {

        // Get video URL from the DB and stream the video

        Video video = videoService.getVideoById(videoId);

        if (video == null) {

            return ResponseEntity.notFound().build();

        }


        Resource videoStream = videoService.getVideoStream(video.getStreamUrl(), codec, resolution,
offset);


        return ResponseEntity.ok()

                .contentType(MediaType.valueOf("video/mp4"))

                .body(videoStream);
    }

}
```

4. Tagging and Search Service - For Searching Videos Based on Tags and Titles

Tag Model:

```
@Entity

public class Tag {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;



    private String name;



    @ManyToMany(mappedBy = "tags")

    private List<Video> videos;



    // Getters and Setters

}
```

Search Service (with Elasticsearch):

```
@Service

public class SearchService {



    @Autowired
    private ElasticsearchRepository<Video, Long> videoElasticsearchRepository;


    public List<Video> searchVideos(String query) {

        return videoElasticsearchRepository.findByTitleContainingOrTagsNameContaining(query,
query);

    }

}
```

5. Comment Service - Handling Comments on Videos

Comment Model:

```
@Entity

public class Comment {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;


    private String commentText;


    @ManyToOne

    @JoinColumn(name = "video_id")

    private Video video;


    @ManyToOne

    @JoinColumn(name = "user_id")

    private User user;
    // Getters and Setters

}
```

Comment Service:

```
@Service

public class CommentService {



    @Autowired

    private CommentRepository commentRepository;



    public Comment addComment(Long videoId, Long userId, String commentText) {

        Video video = videoRepository.findById(videoId).orElseThrow();

        User user = userRepository.findById(userId).orElseThrow();



        Comment comment = new Comment();

        comment.setVideo(video);

        comment.setUser(user);

        comment.setCommentText(commentText);



        return commentRepository.save(comment);

    }

}
```

6. Video Processing Pipeline - Video Chunking and Transcoding

Video Processor:

```
@Service
public class VideoProcessor {

    @Autowired
    private S3Service s3Service;

    public void processVideo(byte[] videoData) {
        // Chunking, Transcoding, Quality Conversion
        byte[] transcodedData = transcodingService.transcode(videoData);
        s3Service.saveTranscodedData(transcodedData);
    }
}
```

7. S3 Service - For Storing and Retrieving Video Data

S3 Service for Uploading and Retrieving Videos:

```
@Service
public class S3Service {

    private AmazonS3 amazonS3;
    private String bucketName = "your-s3-bucket-name";

    @Autowired
    public S3Service(AmazonS3 amazonS3) {
        this.amazonS3 = amazonS3;

    }



    public String uploadVideo(byte[] videoData) {

        String videoKey = UUID.randomUUID().toString();

        amazonS3.putObject(bucketName, videoKey, new ByteArrayInputStream(videoData), new
ObjectMetadata());

        return amazonS3.getUrl(bucketName, videoKey).toString();

    }



    public Resource getVideoStream(String videoUrl) {

        return new UrlResource(videoUrl);

    }



    public void saveTranscodedData(byte[] transcodedData) {

        String videoKey = UUID.randomUUID().toString();

        amazonS3.putObject(bucketName, videoKey, new ByteArrayInputStream(transcodedData), new
ObjectMetadata());

    }

}
```

8. Analytics Service - For Metrics Collection (Example for Video Views)

Video Views Model:

```
@Entity
public class VideoView {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    @JoinColumn(name = "video_id")
    private Video video;

    @ManyToOne
    @JoinColumn(name = "user_id")
    private User user;

    private LocalDateTime viewTime;

    // Getters and Setters
}
```

Analytics Service:

```
@Service
public class AnalyticsService {
    @Autowired
    private VideoViewRepository videoViewRepository;

    public void recordVideoView(Long videoId, Long userId) {
        Video video = videoRepository.findById(videoId).orElseThrow();
        User user = userRepository.findById(userId).orElseThrow();

        VideoView videoView = new VideoView();
        videoView.setVideo(video);
        videoView.setUser(user);
        videoView.setViewTime(LocalDateTime.now());

        videoViewRepository.save(videoView);
    }
}
```

Final Thoughts:

This is a high-level Java implementation of the Netflix-like streaming service. It covers all major components like video upload, streaming, search, user management, comments, and analytics. Additionally, the system is designed to scale, and services communicate through REST APIs (you can also opt for gRPC for better performance).

To ensure high availability and reliability, we can consider using cloud services (AWS, GCP, etc.), Content Delivery Networks (CDN), and horizontal scaling of microservices.