Distributed Systems (41720)

Team : Latecomers Project : Clip Cut

Yash Singh Pathania `<24204265>`
Ryan Jeffares `<23201777>`
Euan Leith `<24108821>`
Nishal Koshy Philip `<24241487>`
Sinem Taskin `<24283182>`

Synopsis

We set out to build a distributed system that handles video uploads, transcoding, subtitle generation, and user authentication, all while offering fast and resilient services to end-users. The system involves multiple microservices, each performing specific tasks such as user authentication, video uploading, video processing (in different resolutions), audio transcription via Whisper AI (that runs locally in cluster even leverages Gpu If you have one) , and monitoring.

Application Domain

The application domain is essentially an online video hosting and streaming platform. Users can register, log in, upload their videos, and watch them. Videos are transcoded into different resolutions for adaptability, while Whisper AI automatically generates subtitles (or full transcripts).

What the Application Does

1. **User Registration and Login**

    - Facilitated by a dedicated `user-service` connected to a **Postgres** database for secure storage of user credentials.

2. **Video Upload and Storage**

    - Handled by the `video-upload-service`, which stores the original video in **MongoDB (GridFS)** for efficient handling of large media files.

3. **Distributed Message Passing**

- After a successful upload, a message is published to **Redis** using its built-in pub/sub mechanism, notifying the `video-processing-service` that a new video is ready for processing.

4. **Video Processing and Transcription**

    - The `video-processing-service` then retrieves the video from MongoDB, transcodes it to various resolutions, and calls the `audio-service` (which uses Whisper AI) to generate subtitles.

5. **Monitoring and Fast Retrieval**

    - A `monitoring-service` maintains a **Postgres** cache of processed videos for quick lookups, while the raw media remains in MongoDB. The monitoring service is notified of any video status changes via lightweight HTTP updates from the other services.

Overall, this distributed architecture ensures that each service is focused on a well-defined responsibility, facilitating scalability, resilience, and modularity.

Technology Stack

Below are the primary technologies we used, along with why we chose them:

1. **MongoDB (GridFS)**

    - **What It's Used For**: Storing large video files (raw uploads and processed outputs).
    - **Why We Chose It**: GridFS offers an easy way to chunk and store large files without requiring a rigid schema. This is ideal for raw and transcoded videos.

2. **Postgres**

    - **What It's Used For**: The `user-service` stores user credentials and session info here. The `monitoring-service` also uses Postgres to keep track of processed video metadata for fast queries.
    - **Why We Chose It**: We needed a reliable, relational database for structured data like user accounts and for indexing metadata. Postgres excels in data integrity and complex queries.

3. **Redis**

- **What It's Used For**: Operating as a pub/sub channel for distributing notifications between the `video-upload-service` and the `video-processing-service`.
- **Why We Chose It**: Redis pub/sub is straightforward, fast, and reliable for lightweight messaging, ensuring near real-time communication about new uploads.

4. **audio-service (Whisper AI)**

    - **What It's Used For**: Generating video transcriptions/subtitles by extracting audio and running Whisper AI.
    - **Why We Chose It**: Whisper AI provides robust speech recognition, greatly enhancing accessibility of uploaded videos. Its integration as a microservice also keeps the architecture modular.

5. **frontend**

    - **What It's Used For**: A simple user interface for uploading videos, logging in/out, and viewing streaming content.
    - **Why We Chose It**: Having a dedicated frontend service keeps the client logic separate, making it easy to iterate UI/UX changes independently.

6. **user-service**

    - **What It's Used For**: Authenticates users (login/registration) and retrieves processed videos that belong to each user.
    - **Why We Chose It**: Decoupling user logic from other services helps maintain a clear separation of concerns, reduces complexity, and improves security.

7. **video-processing-service**

    - **What It's Used For**: Receiving notifications from Redis, transcoding the video into various resolutions, and invoking the audio-service for subtitles.
    - **Why We Chose It**: Splitting transcoding logic into its own service allows better resource scaling. We can spin up multiple instances if the load gets high.

8. **video-upload-service**

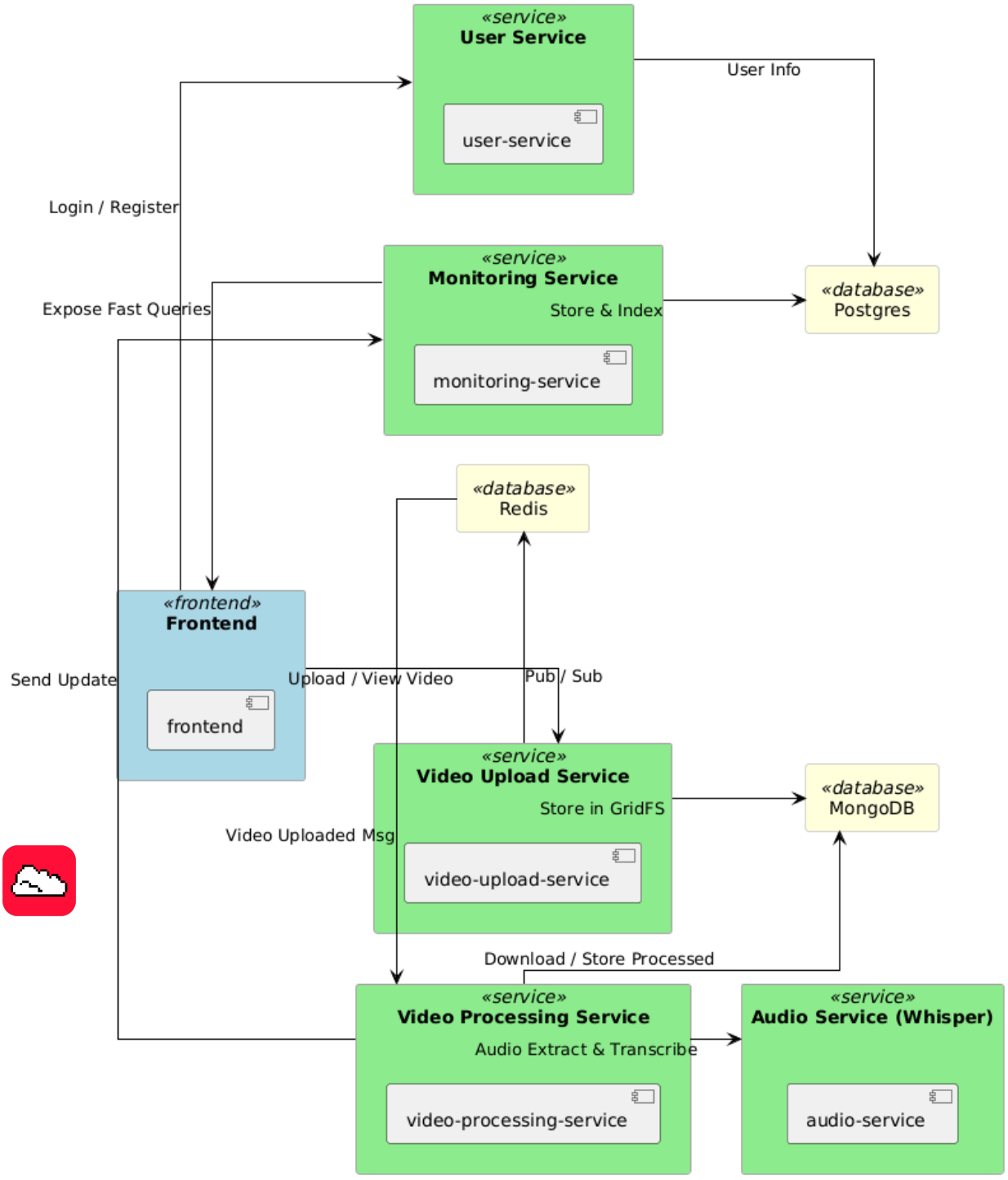
    - **What It's Used For**: Handles uploading videos from the frontend, stores them in MongoDB, and publishes a "new video" event to Redis.
    - **Why We Chose It**: This service isolates file upload concerns, ensuring that the system remains modular and each part can be scaled independently.

9. **monitoring-service**

- **What It's Used For**: Tracks system updates via lightweight HTTP notifications and stores metadata in Postgres for quick indexing.
- **Why We Chose It**: We needed a centralized place to gather metrics and statuses from all services so that queries (e.g., "Get all videos for user X") can be quickly answered without scanning large media files.

System Overview

To illustrate the interactions among these components, here's a high-level architecture diagram



How the System Works

1. **User Registration and Login**

    - The `frontend` sends credentials to the `user-service`, which verifies against Postgres.
    - On success, the `user-service` returns a session token or similar credential to the client.

2. **Video Upload**

    - Once logged in, a user uploads a file via the `video-upload-service`, which chunks and stores the raw file in MongoDB (using GridFS).
    - The service then publishes a message to Redis indicating a new video is ready for processing.

3. **Video Processing**

    - The `video-processing-service` is subscribed to the Redis channel and is notified of new videos.
    - It fetches the file from MongoDB, transcodes it (e.g., multiple resolutions), and calls the `audio-service` to generate subtitles via Whisper AI.
    - The results (transcoded files + subtitle files) are placed back into MongoDB.

4. **Monitoring and Indexing**

    - The `video-processing-service` sends a lightweight HTTP request to the `monitoring-service` whenever a processing milestone is reached (e.g., video conversion done).
    - The `monitoring-service` updates a Postgres index so that user queries about videos can be quickly answered.

5. **Video Consumption**

    - When a user wants to watch a video, the `frontend` fetches the relevant links from MongoDB (potentially via the `video-upload-service` or a dedicated streaming endpoint).
    - Subtitles are provided on the fly from the processed data.

Scalability and Fault Tolerance

- **Microservices**
  Breaking the system into microservices allows each part to be scaled independently. If user uploads spike, we can scale the `video-upload-service`. If transcoding jobs back up, we can scale the `video-processing-service`. Most of this is handled very well with Kubernetes, which is built to manage such scaling automatically. For the

`video-upload-service` and `video-processing-service`, we have also used the Horizontal Pod Autoscaler (HPA), which allows the system to automatically adjust the number of running pods based on the CPU usage or other specified metrics. This ensures that during peak times, when the load increases, more pods are automatically added to handle the load, and during quieter times, the number of pods is reduced to save resources. This dynamic scaling capability helps in maintaining optimal performance and cost efficiency, ensuring that the services can handle varying loads without any manual intervention. Additionally, the use of Kubernetes and HPA aligns well with our microservices architecture, promoting high availability and fault tolerance across all services.

- **Redis Pub/Sub**
  The asynchronous communication model prevents blocking calls between upload and processing. If the `video-processing-service` is busy, incoming uploads won't fail; they'll queue in Redis until a consumer is ready.

- **Redundancy**

  - **MongoDB** is set up with a replica set for high availability.
  - **Postgres** is setup with read replicas .
  - **Redis** is clustered for failover.

- **Graceful Failure**
  Even if the `video-processing-service` goes down, uploaded videos remain safe in MongoDB, and once the service restarts, it will process the pending jobs from Redis.

Contributions

Yash Singh Pathania <24204265> Yash was responsible for developing the frontend and user-service, handling user authentication and integrating PostgreSQL for credential storage. He also managed MongoDB's GridFS for the video-upload-service, ensuring efficient video file handling and integration with Redis for messaging. Additionally, Yash implemented the overall system architecture and the initial setup, including Kubernetes with Horizontal Pod Autoscaler (HPA) for scalable deployment.

Ryan Jeffares <23201777> Ryan engineered the video-processing-service, focusing on video transcoding using FFmpeg and coordinating transcription services with the audio-service using Whisper AI.

Euan Leith <24108821> Euan managed the monitoring-service, setting up PostgreSQL schemas for video metadata and implementing lightweight HTTP endpoints for timely service updates.

Nishal Koshy Philip <24241487> Nishal assisted in developing the video-upload-service, focusing on backend logistics for video storage and ensuring seamless integration with Redis for real-time messaging.

Sinem Taskin <24283182> Sinem contributed to enhancing the frontend and user-service, improving user interface components for video viewing and interaction. She also worked on implementing Swagger for API documentation, ensuring a clear and user-friendly interface for API interactions.

Reflections

Key Challenges and How We Overcame Them

1. **Managing Large Video Files**
    - The biggest challenge was dealing with large file uploads and making sure the system was resilient to spikes in requests. We tackled this by storing data in MongoDB GridFS, which gracefully handles large files in chunks.
2. **Coordinating Multiple Services**
    - Synchronizing the `video-upload-service`, `video-processing-service`, and `monitoring-service` required a clear messaging pattern. Redis pub/sub helped decouple services, so the uploader doesn't have to wait for the processing to finish.
3. **Ensuring Reliability and Scalability**
    - We overcame potential single points of failure by adopting microservices that can each be scaled horizontally. For instance, running multiple instances of the `video-processing-service` let us handle more conversion tasks in parallel.
4. **Subtitles Generation (Whisper AI)**
    - Integrating Whisper AI for transcriptions meant we had to carefully manage audio extraction and synchronous calls between the `video-processing-service` and the `audio-service`. Thorough testing and structured gRPC-like calls (or REST) ensured stable performance.

What We Would Do Differently

1. **Full Event Streaming**

- If we had more time, we might switch to a more robust system like Apache Kafka, which offers persistent queues and better fault tolerance than Redis pub/sub for job retention. This would handle replay scenarios if the processing service were down for longer.

2. **More Thorough Monitoring**

    - A comprehensive monitoring stack (Prometheus + Grafana) would offer deep insights into performance metrics, memory usage, and throughput. We currently rely on logs and simple database checks.

3. **Adaptive Streaming**

    - While we transcode into multiple resolutions, implementing a truly adaptive streaming protocol (e.g., HLS or DASH) could further improve user experience by automatically selecting the best resolution based on network conditions.

Lessons Learned About the Technologies

- **MongoDB GridFS** is well-suited for large-file storage, but we need to carefully manage chunk size for optimal performance.
- **Redis** provides simple and lightning-fast pub/sub functionality, though it lacks persistent queues out of the box.
- **Postgres** is excellent for structured data and indexing, but we must stay on top of schema management.
- **Microservices Architecture** truly decouples components, but demands careful planning of inter-service communication, especially for concurrency and error handling.

Limitations and Benefits

- **Limitations**:

    - If a service fails (e.g., `video-processing-service`), new messages might accumulate in Redis. If it's down for an extended period, we can risk data inconsistency unless we implement fallback or re-queue logic.
    - MongoDB performance might degrade if we scale to extremely large volumes, so we might consider sharding or a more advanced storage approach if usage grows massively.

- **Benefits**:

    - Highly modular design: Each service focuses on a distinct responsibility, making the system easier to maintain and scale.
    - Better user experience: The system processes videos asynchronously, **so users can continue uploading content without waiting.**
    - Fault tolerance: Thanks to the microservices approach, the entire system rarely goes down if one component fails.

THANK YOU

HAVE A GREAT DAY