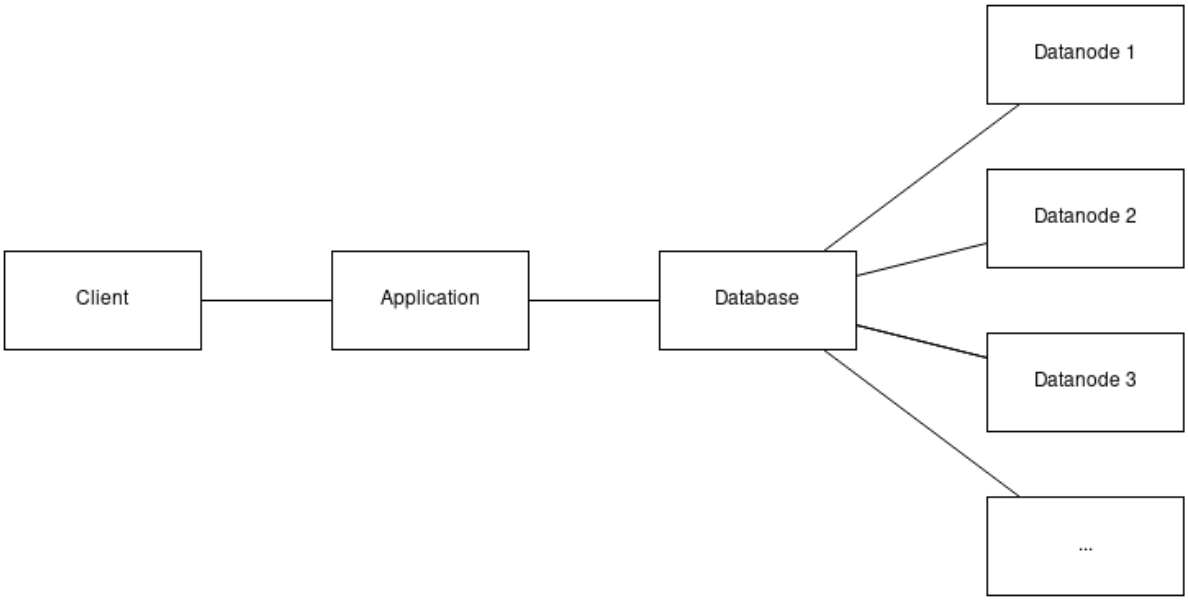
MyYoutube - Entrega 2

Gustavo Ribeiro Moura

A arquitetura do sistema é composta por quatro entidades principais: Cliente, Aplicação, Banco de Dados e Nós de Dados.
O Cliente refere-se ao navegador usado pelo usuário para acessar a aplicação.
A Aplicação é responsável por toda a lógica de negócio e tem acesso ao Banco de Dados.
O Banco de Dados abstrai o armazenamento de dados e possui conexão com vários Nós de Dados, responsáveis pelo armazenamento em si. Um servidor de Banco de Dados coordena e abstrai o acesso a esses nós. O índice distribuído é mantido no servidor do Banco de Dados.



Application

Endpoints:

1. **GET /video/{video_id}**
    - Retrieves a video file based on its `video_id` from the database.
    - Sends a request to the database to get the video file and streams it back to the client.
2. **GET /list**
    - Requests the list of available videos from the database.
    - Generates an HTML page displaying the list of videos and sends it back to the client.
3. **GET /watch/{video_id}**
    - Fetches metadata for a specific `video_id` from the database.
    - Generates an HTML page for watching the video player with its metadata and sends it to the client.
4. **GET /search?query={search_query}**
    - Initiates a search for videos based on the `search_query`.
    - Searches the database for matching videos, generates an HTML list of matched videos, and sends it back to the client.
5. **GET /delete/{video_id}**
    - Deletes the video associated with `video_id` from the database.
    - Initiates a request to delete the video and retrieves an updated list of videos, then generates an HTML list and sends it to the client.
6. **GET /favicon.ico**
    - Retrieves the favicon file.
7. **GET /**
    - Retrieves the index page.
    - Generates an HTML index page and sends it to the client.
8. **POST /upload**
    - Handles video uploads to the server.
    - Expects a `multipart/form-data` request containing the video file.
    - Sends the video file to the database for storage and generates an HTML page for the uploaded video.

Behavior:

- The server listens on a specified port (`PORT = 8080`) for incoming connections.
- Upon connection, a new thread is spawned to handle the HTTP request.
- The server uses a custom HTTP parser to interpret incoming requests and responses.
- Depending on the HTTP method (GET or POST) and the requested URL, different actions are taken:
    - GET requests:
        - Handle video retrieval, list retrieval, video watching, search, deletion, favicon retrieval, or index page retrieval.
    - POST request:
        - Handle video uploads by sending the uploaded video file to the database for storage.

Interaction with Database:

- Communicates with a database server running on `localhost:8081` for various operations like getting lists, metadata, file retrieval, deletion, and upload.

Response Handling:

- Responds with appropriate HTTP headers and content types based on the requested operation and data type.
- Generates HTML responses for various operations such as video list, video player, search results, or error (404) pages.

This HTTP API essentially allows clients to interact with the server for video-related operations like uploading, viewing, searching, and deleting videos, leveraging a custom server-to-database communication protocol over HTTP.

Database

Endpoints:

1. **GET /file**
    - Downloads a file with a specific `id` from the distributed system.
    - Sends the requested file to the client.
2. **GET /list**
    - Retrieves a list of available files stored in the distributed system.
    - Sends the list of files to the client.
3. **GET /metadata**
    - Retrieves metadata for a specific file based on its `id`.
    - Sends the metadata information to the client.
4. **GET /delete**
    - Deletes a file based on its `id` from the distributed system.
5. **POST /upload**
    - Handles file uploads to the distributed system.
    - Initiates the file upload process and sends the file to multiple datanodes for storage.

Behavior:

- The script maintains a global metadata dictionary to store file metadata.
- It interacts with a set of datanodes listed in the `datanodes.txt` file for file storage and retrieval.
- Utilizes a custom HTTP parser to interpret incoming requests and responses.

File Operations:

- **Upload**:
    - Generates a unique ID for the file and stores its metadata.
    - Utilizes a replication factor of 3 to upload the file to multiple datanodes.
    - Updates metadata with the list of datanodes containing the file.
- **Download**:

    - Retrieves a file from a randomly selected datanode based on metadata.
- **Delete**:
    - Removes a file from all datanodes and deletes its metadata.

Communication with Datanodes:

- **Upload**:
    - Sends file data to datanode for storage.
- **Download**:
    - Requests file data from a datanode.
- **Delete**:
    - Requests deletion of file data from a datanode.

Metadata Operations:

- Handles metadata updates during upload, download, and deletion of files.

This API essentially allows clients to interact with a distributed file system for file management operations such as uploading, downloading, listing, and deleting files, leveraging a distributed architecture to store and retrieve files across multiple datanodes.

Datanode

Endpoints:

1. **GET /file**
    - Downloads a file with a specific `id` from the local file storage.
    - Sends the requested file to the client.

2. **GET /delete**
    - Deletes a file based on its `id` from the local file storage.
3. **POST /upload**
    - Handles file uploads to the local file storage.
    - Expects a multipart/form-data request containing the file data.
    - Stores the uploaded file locally.

Behavior:

- The script maintains a folder named "files" to store uploaded files.
- Utilizes simple file I/O operations for file upload, download, and deletion.

File Operations:

- **Upload**:
    - Writes chunks of the uploaded file to a local file based on the `id`.
- **Download**:
    - Reads chunks of a file from the local file storage based on the provided `id`.
- **Delete**:
    - Removes a file from the local file storage based on the provided `id`.

Handling HTTP Requests:

- Listens for incoming connections on a specified port (`PORT = 8082`).
- Spawns a new thread to handle each incoming HTTP request.
- Uses a custom HTTP parser to interpret incoming requests and responses.

This API is a basic implementation for local file management through HTTP endpoints, allowing clients to upload, download, and delete files from a local storage system.