MCFlix: An Android application for live streaming and watching VoD content

Rúben Carvalho Faculty of Science, University of Porto

In this small report I describe the overall structure of this project and the main difficulties I faced, coupled with the solutions I decided to use to overcome these problems.

1 Architecture

This application has two main architecture components: the **client** and the **public cloud**. An application server that sits on the cloud serves as the unique point of communication between the cloud system and the client. The communications between client and the application server are done through the HTTP/Rest API.

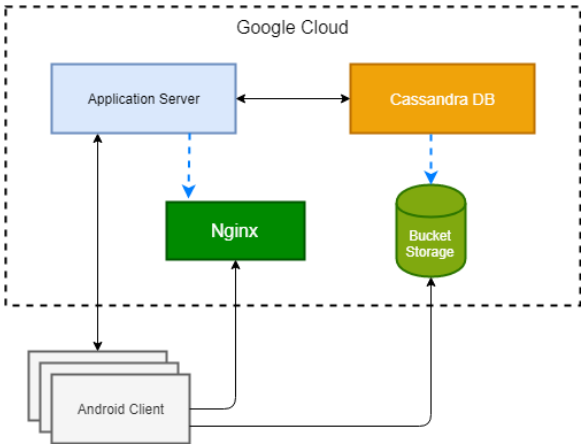

Figure 1: MCFlix platform architecture.

In the figure above, blue arrows represent references and black arrows represent connections.

I now explain the purpose of each architecture element and how it communicates with other elements in this model.

Android Client

The client represents the Android Application that sits on a Java implementation of the different activities used to communicate with the server to retrieve the data related to media content.

The application can be divided in three main activities: *Watching VoD content*, *Watching Streaming content* and finally *Streaming your own content*. These are displayed in the main screen of the application.

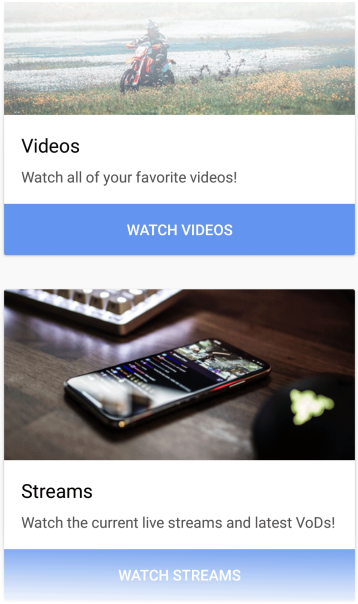

Figure 2: Main activity screen.

Public Cloud

The public cloud of the system is made up of the Application Server, a Database and a livestreaming server.

Application Server

The application server is the main central point of the back-end of the system. It provides a simple REST API

that allows the clients to connect to and query for content and livestream links. It is implemented in Java.

API methods:

VoD Related

- **GET** /movies/categories - *Retrieve all available video content categories as a list.*
- **GET** /movies/search **PARAMETERS** category - *Retrieve all available video content for a given category as query parameter.*

Streaming Related

- **GET** /stream/streams - *Retrieve all streams that are live at the moment as a list.*
- **GET / PARAMETERS** id - *Given a stream id as a query parameter, returns the url of that stream.*
- **POST / PAYLOAD** Stream Request - *Given a stream request with id, tries to create the given stream.*
- **POST** /delete **PAYLOAD** Stream Request - *Given a stream request with id, tries to delete the given stream.*

It is important to note how I could have created a DELETE method at the root path for the Streams. I decided to change to the POST to /delete because the http requester I used for our client application (Android Volley) did not support DELETE requests with a body.

Because of time constrains I was not able to fix this anomaly.

Database

The database stores metadata related to the videos, such as a variety of links with different encodings for each video. These links point directly to a Google Cloud Bucket, where the files are stored.

For this project I decided to use **Apache Cassandra** for the database. In this way I can avoid the added complexity and performance downgrade when compared to querying a relational database.

The schema I decided to create to store the video content is defined as such:

mcflix.movies schema
movie_id int PRIMARY KEY
category set<text>
length_seconds int
name text
url map<text,text>
year int

Livestreaming Server

As for the livestreaming server, I used Nginx as a reverse proxy hosting an rtmp service. The rtmp configuration was simple. I added the rtmp module when installing nginx and added the following lines in the **nginx.conf** file.

```
rtmp {
    server {
        listen 1935;
        chunk_size 4096;

        application live {
            live on;
            record off;
        }
    }
}
```

When a user wants to watch a certain livestream, the client requests a certain livestream link to the application server and connects directly to the Nginx server after.

Bucket Storage

The video content is all stored on a Google Cloud Bucket. The database references these public links for the client to access directly.

2 Challenges and Solutions

Debugging

The first challenge was to find a way of quickly building and debugging the android application and the application server coupled with the database and streaming server in a quick and easy way.

I decided to host all the server applications in the local network and use two physical android devices to test all the communication to the server, later deploying the system when the development phase was complete.

The reason I needed two devices and not only one was for testing the livestream creation and viewing.

Content viewing and creation

Before starting the implementation of the VoD and streaming content viewing, I first had to define what functionalities and general design I wanted to create.

As proposed by the professor, I defined three different activities: Watching VoD, watching streams and creating your own stream.

VoD

I implemented a category selector that requests the application server to query the database for all the available categories and displays them as a selector to the user to pick from.

After the user selecting the category he intends to watch, the client requests a query of all the available content to the application server for the clicked category and displays all the content using a custom adapter implemented by myself.

The implementation of this adapter can be consulted in the **ContentAdapter** class.

How to play the VoD content on the devices?

In order to play the content in each device I decided to call a web view client from the android application and use the already implemented video player on the web view client to play the VoD on the devices, feeding it simply the url of the content that is in the response given by the application server.

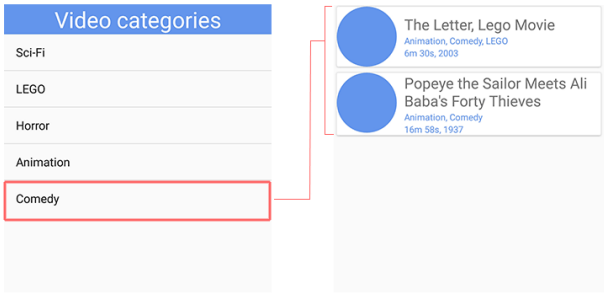

Figure 3: Video category selector and content adapter display.

Watching streams

In order to play RTMP streams on the client I followed an already existent implementation [4] of a simple player based on the ExoPlayer library [2].

Streaming content

In order for the user to stream his content, I used a dependency that implements a simple-to-use Rtmp Publisher for the Android platform [3]. This provides a simple API that allows the connection of the camera and microphone of the device (if permissions are given) to the publisher of RTMP that connects to the livestreaming nginx server I provide.

On top of that I implemented a simple interface to publish and stop publishing a stream and select the title of the stream.

Application Server

It is important to note that the application server, the database and nginx server are all hosted on the same machine running Linux Ubuntu. This machine was at first the development workstation and later the google cloud virtual machine where I deployedthe system.

Figure 4: Streaming activity.

Connecting to the Cassandra database

In order to access the local cassandra database from the application server, I used the implemented drivers by Datastax to communicate with Apache Cassandra [1].

Managing the state of the current online streams

The simple idea I came up with to solve this issue is to store the information of the current online streams in the state of the application server (using @Context to store and retrieve the current streams), as I believe it is not important to store persistent information in the scope of this project.

Deployment

The deployment process was straightforward. I changed the relevant ips on the code to the VM external static one and installed all the required dependencies on the google cloud virtual instance running Ubuntu 16.04 LTS.

For the application server I built a **shaded uber jar** with maven that includes all the relevant libraries linked. I uploaded this shaded jar file to the server and simply run it with the java -jar command.

Since I had stored all the relevant configurations before for the Nginx and Cassandra instances, in the deployment phase I just uploaded them to our cloud instance and the server was up and running.

To let it running as a background process when I close the SSH connection I used the **screen** utilitary.

3 Future development

I was deeply invested in the development of this system for curricular purposes. As such, I can not help but to mention a few development features that could be added in the future if I intend to continue working on our project. These are listed next in descending order of importance.

Software Testing

I am a firm believer that testing is one of the most important methods of assuring this project would be officially deployed with the least number of bugs possible. While testing was not required for the curricular scope of this project, I am sure that the implementation of unit and integration tests would greatly benefit the development of this application.

Authentication

While this was first proposed by the professor, to implement Google Auth 2.0 in our system, unfortunately the time constraints did not allows me to do so. I believe this is highly important for the managing of privileges on the application and also to guarantee the new GRPD regulations necessary to launch the system to the users.

This could also extend to the data mining of knowledge related to movie and video recommendations by user, because at the moment the system has no method of identifying which user is watching which content.

Rebroadcasting of older streams

Another important feature I believe is focal when creating a livestreaming service is the ability to watch older broadcasts. For this matter, I need an infrastructure that

supports large amounts of data storage on the nginx server side.

As far as I researched, the process of recording in itself is as straightforward as simply enabling the recording option that is currently disabled in the nginx properties and specifying the directory in which to store the content.

Rating system

Following the train of thought of the data mining of knowledge about the content available to each user, a good method to suggest new content to users would be to predict their ratings. For this matter, a learning algorithm would have to base this prediction in the already existing ratings of a certain user or groups of users.

Because of this, I believe that it would be beneficial to implement a simple star or number rating system in the application. The ratings would have to be stored in an additional relational database system accompanied by user account information.

4 Comments and conclusions

Above all, I want to thank the professor for the idea of this project. While I was definitely not prepared in the beginning for a project of this size and failed to implement the authentication part of the project, it was extremely beneficial to understand how all the components of a system need to be built together to deploy an android application.

I also want to point out the necessity of having to read the code of certain dependency implementations in order to understand how certain APIs that did not have any documentation worked. I believe this project helped developing a plethora of good habits in me as a software engineer.

References

[1] Datastax. Java driver for apache cassandra. https://github.com/datastax/java-driver.

[2] github google. Exoplayer. https://github.com/google/ExoPlayer.

[3] github TakuSemba. Android: Rtmp publisher. https://github.com/TakuSemba/RtmpPublisher.

[4] github teocci. Android: Rtmp player. https://github.com/teocci/Android-RTMP-Player.