FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Low-Cost Transcoder

Luís Filipe Rodrigues Coelho

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Maria Teresa Andrade

Second Supervisor: Nuno Mota

February 24, 2020

Low-Cost Transcoder

Luís Filipe Rodrigues Coelho

Mestrado Integrado em Engenharia Informática e Computação

February 24, 2020

Abstract

Multimedia is presented nowadays in almost every device with big costs associated with it, making its processing very difficult and device-dependent, some devices can decode and play certain video streams, others cannot. This is a problem in IP based networks for media content broadcast, where it's not feasible to send multiple media profiles, instead, a single video signal with a specific format is broadcast to reach every device and thus, making it a challenge to guarantee a single media profile will be playable across all devices.

That barrier can be addressed by changing the characteristics of the original media content into something more acceptable by the majority of clients. That process is called transcoding and this thesis' work tries to mitigate one of the biggest problems involving transcoding operations: cost. This thesis is focused on media content transcoding for IPTV systems and was proposed by Nonius, a hospitality focus company, to reduce the costs of their current solution.

Our proposed solution uses an ARM-based board, Android Set-Top box, with decoding and encoding hardware assisted operations, known for their low cost and high capabilities of video processing due to its dedicated media processing unit. The system can perform live media content transcoding of several codecs, containers, and sources, and outputs its content to IP networks. The achieved results show that, although the solution is highly dependent on the hardware capabilities, it is possible to guarantee savings up to 80% when compared with other market solutions without abdicating too much quality using real-world scenarios.

With the proposed solution, Nonius IPTV solution will not only be able to reach legacy-hardware playback consumers but will also allow to reduce the costs during and after transcoding due to the support of new codecs.

Acknowledgements

I want to thank my supervisor Maria Teresa Andrade, for the knowledge she shared with me, and for all the dedication in answering my questions and help me improve this dissertation.

I would also like to thank my supervisors at Nonius, Nuno Mota, and Décio Macedo, for all the patient, guidance and availability to guide me during the development process which made this commitment easier.

Additionally, I would like to thank my co-workers at Nonius for the wonderful experience and company that made all this process more enjoyable.

To my friends, and especially my family, for their constant support.

And last but not least to my girlfriend Carolina Vieira, for all the dedication and support in helping me complete this thesis.

Luís Filipe Rodrigues Coelho

"Many of life's failures are people who did not realize how close they were to success when they gave up."

Thomas Edison

Contents

CONTENTS

List of Figures

LIST OF FIGURES

List of Tables

Abbreviations

API	Application Programming Interface
AV1	AOMedia Video 1
AVC | H264	Advanced Video Coding
CPU	Central Processing Unit
DASH	Dynamic Adaptive Streaming over HTTP
DCT	Discrete Cosine Transform
DPCM	Differential Pulse Code Modification
DSP	Digital Signal Processor
HD	High Definition
HEVC | H265	High Efficiency Video Coding
HLS	HTTP Live Streaming
HTTP	Hypertext Transfer Protocol
IPTV	Internet Protocol Television
MPEG	Moving Picture Experts Group
MPEG-TS	MPEG transport stream
OpenCL	Open Computing Language
OTT	Over The Network
PCR	Program Clock Reference
PID	Packet Identifier
PMS	Property Management Systems
PSNR	Peak signal-to-Noise Ratio
SD	Standard Definition
STB	Set-Top Box
RGB	Red, Green and Blue
URI	Uniform Resource Identifier
VLC	Variable Length Coding
VoD	Video-on-Demand

Chapter 1

Introduction

1.1 Context / Background

Multimedia is one of the most important aspects of the internet and one of the most expensive computation solutions. With the emerging requirements from clients, several protocols were developed in an attempt to reach the biggest possible number of consumers with the best possible quality. One of the main techniques to satisfy this objective is to offer a large number of different profiles for the same service and let the client choose the one that better suits its network and computation limitations. This is a relatively recent technique and is mostly embraced by mobile devices.

Another technique, and probably the most common and the main focus of this thesis research, is the distribution of a single media content profile that suits most of the clients. This technique is widely use to bring TV to the consumer's house or in IPTV systems where the consumer capabilities are well known and one profile can easily be selected to serve them all 1.1.

However, with the continuously increase in demand for multimedia content and the high complexity and costs to perform it, becomes necessary to apply those techniques in a cheaper way to mitigate the problems arose from "one profile serves them all".

In order to provide digital content, many content providers offer a Tv Box that is connected to client's TV to reproduce all available content. This scenario works fine if the client only has a few TVs however, in other scenarios with dozens of rooms and multiple TVs, it is not feasible for the client, and neither for the provider, to use one TV Box for each TV.

TVs nowadays are capable of performing the same operations of a TV Box however, it's simpler to distribute a service that works in a specific hardware than implementing it for several different hardware architectures and performance capabilities. But, for hotels, this system is expensive so, instead of having a TV connect to a TV Box, most of the TVs are not connect to any Box because they are able to perform those operations. Unlike the TV Box setup, the content is

multiplexed and broadcast to all TVs and, unlike the first mentioned technique, only one profile is offered to reduce bandwidth.

In those scenarios, one can't simply assume the client will be able to consume the media content due to its characteristics therefore, it becomes necessary to transform the media content format into something more suitable to all clients.

This thesis investigation was proposed by Nonius, a tech company dedicated to hospitality technology development. Nonius provides IPTV solutions for hotels and faces this problem frequently. Since they have to support old and new TVs and different content providers, it's hard to guarantee that each TV supports all the providers' content. Changing the characteristics of the media stream becomes necessary and this thesis aims to evaluate a cheaper solution to do it.

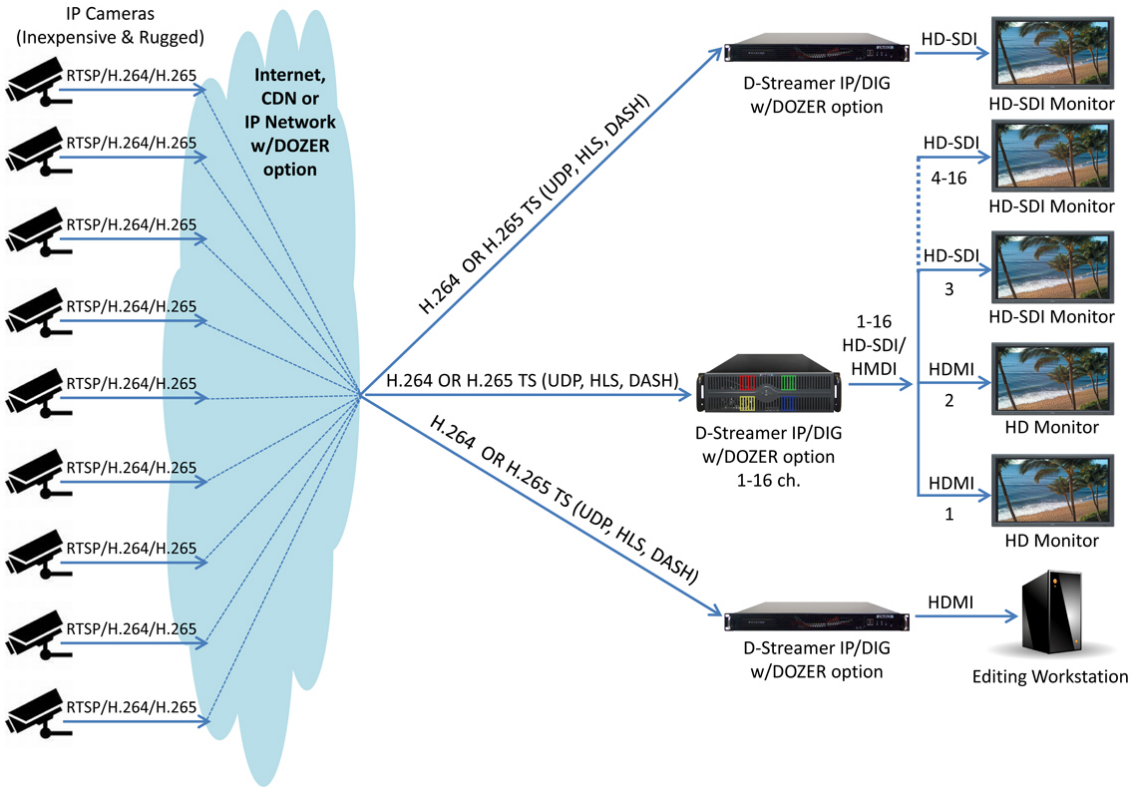


Figure 1.1: IPTV overview [dve20]

.

1.2 Aim and Goals

In order to overcome the problem specified in the last section, Nonius came up with the idea to create a transcoder to standardize the services' characteristics across different providers. Since the transcoder is able to also change the format, one can change the media content format into something that is supported by most of the available hardware characteristics in a specific facility.

This thesis consists of a standalone system capable of performing transcoding operations of at least a single service and distribute it in a multicast network. In order to reduce cost per service (channel), this project will be implemented in an Android Set-Top Box platform.

One of the problems that arise is concerned with the fact that low budget Android STBs are not designed for content distribution and processing even though, they are capable of performing it. This project consists, therefore, in developing the necessary procedures to perform the desired operations of consuming, transforming and distributing media content.

The main motivation behind this project is to offer all available services to the consumers, eliminating some small barriers due to different hardware capabilities. This kind of solution are commonly but at a high cost and knowing that a cheap Android STB can reproduce a 4K video, made us wonder if they are capable to be in charge of transcoding a given service. The main objectives to reach that vision are:

- **Transcoder Implementation**. Procedure capable of taking one media content as input and transform it according to user desired specifications.
- **Service Broadcast in a Multicast Network**. In order to use the solution in IPTV networks.
- **Quality Assurance at Low Cost**. To maintain the highest quality possible to fulfill current client expectations.

1.3 Dissertation Structure

This document is structured in seven chapters:

1. **Introduction:** chapter 1, the project is introduced as well as the motivation, objectives and project's outline
2. **Literature Review:** chapters 2 and 3. In chapter 2 some context literature review is presented to illustrate the technologies behind this thesis research and its impact on decision making. In chapter 3, it's presented the literature review concerned with this thesis transcoder implementation and also the alternatives and their inconveniences.
3. **Specification of the proposed solution:** chapter 4 presents the main problem of this project, the possible solutions and the specification of the solution chosen.
4. **Implementation Specification:** chapter 5 presents the project implementation and the challenges faced.
5. **Obtained Results and Discussion:** chapter 6 presents the use cases of the system and the tests performed.
6. **Conclusion and Future Work:** chapter 7 presents the so far conclusions and the work left to do.

Chapter 2

Video Coding

The evolution and demand for media content distribution have been growing steadily in the past few decades. To comply with the current market requirements, new processes of media encoding and media distribution have been developed however, not all devices support these new procedures.

To overcome these limitations many content providers offer different media profiles to the consumer and, the consumer, chooses the best profile that fits its hardware and channel characteristics. However, even this technique is relatively new, some devices don't support this technique and, in some cases, this technique is not possible due to the scenario limitations.

In IPTV systems, the main focus of this thesis' work, the characteristics of the devices are well known and thus, only one media profile is broadcasted to all consumers. If a consumer does not support the media profile it becomes necessary to change its format otherwise, that consumer will not be able to consume it.

To better understand the procedures and techniques described above, this chapter will start by introducing how an IPTV system works. After this introduction, the chapter will continue by looking at the data being distributed in the IPTV system and introduce the concepts of data encoding and how it's performed. This introduction about the encoding process will allow us to identify how can the data format be changed. Finally, the chapter will overview the most popular protocols of content distribution and, introduce the content of the next chapter.

2.1 IPTV

According to O' Driscoll [kn:07], IPTV is a technology that aims at securely delivering high-quality broadcast television and/or on-demand video and audio content over a broadband network. This method contrasts from traditional terrestrial, satellite or cable television formats and, unlike downloaded media, IPTV offers a continuous media source streaming service and thus, the client can play the content almost immediately.

Since this technology is based on the Internet Protocol, it offers a set of features [VGJ14]. For this thesis' work we can enumerate:

- Instantaneous Playback - The consumer does not have to search the content provider since the media content is already being streamed in the network.
- Personalization - Allows its users to decide what they want to see and when.
- Accessible on multiple devices - Once consumers join the IPTV network they can not only consume the content in multiple televisions but also their mobile devices, PCs and set-up boxes.
- Low bandwidth requirements - Instead of each client download its content and sometimes, the same content is being downloaded across several devices, IPTV, allows to download the content one time and stream it to multiple consumers. Consequently, reducing the Internet bandwidth usage.

According to [LLRC09] there's four domains of IPTV service: :

- **Content Provider**: The entity that owns or is licensed to sell content.
- **Service Provider**: An operator that offers telecommunication services. In this thesis' IPTV scneario, the Content and Service Provider is the Portuguese entities that distribute media content such as MEO or NOS.
- **Network Provider:** The operator that maintains the network components required for IPTV system. In this thesis' work, the network provider is the entity that proposed this investigation, Nonius.
- **End-user**: The actual consumer of the IPTV system media content. In the context of this investigation work, the end-user is the hotel guests.

Many solutions can take advantage of an IPTV solution, and although IP based technologies allow a wide range of interactive TV applications, there are two key services [LLRC09]:

1. **Video on Demand (VoD)**: content is previously retried from the content provider, processed and stored in a local server. The end-user can then select and retrieve the content at any time.
2. **Broadcast digital TV**: a one-way transmission of content from one point to multiple points. The content can either be stored in a local server or live-content retrieved from the provider and immediately broadcast in the network. The end-user has no control to which content is being broadcasted, he can only select which one should be consumed.

This thesis' work is mainly concerned with Broadcast digital TV although, it can support the other IPTV solutions due to the common concepts behind each one. The Nonius IPTV solution is identical to the concepts described, there are two use cases, VoD, and broadcast digital TV. The

objectives illustrated in chapter 1 does not apply to the VoD solution since the available content can be previously processed. However, in live broadcast digital TV scenarios the end-user does not have control of the broadcasted media content and thus, the content must be processed in real-time otherwise, not all end-users will support the media content.

With the scenario explained it is important to investigate how broadcast data can be processed in real-time. Since media content is encoded before distribution or storage, the next section will focus on how data is encoded so that, later conclusions can be taken on how to process it.

2.2 Fundamentals of Compression

2.2.1 Digital Video Overview

Even though the video is perceived for humans as a continuous function in time, it's captured and saved as a discrete function. Video is made by capturing a set of pictures called frames and, when displayed to the human, is perceived as a continuous function in time. The less the interval between consecutive frames the better perception of a time-continuous function.

Frames, on the other hand, are also a discrete function represented by pixels, each pixel containing the information about the color in the picture 2D space. Color information can be represented by different color models [SVT98], the most popular space color and easy to understand are RGB. RGB space color has three components, Red, Green, and Blue, which can be identified as three light sources. Each component is assigned a weight and the sum of all weights (RGB color model is additive) from the three components gives us a space with 16.777.216 colors (when using 256 values to describe each component).

However, YCbCr is the most common color model used, where Y represents the overall brightness called *luma* and Cb Cr components represent the color-difference channel called *chroma*. This model was developed with the human eye in mind since humans are more sensitive to light than colors so, one can use less accuracy in the values of the *chroma* components and for the human eye, the change in color would not be perceived.

Given the mentioned aspects, a video stream can be classified using three characteristics:

- ***frame-rate***: number of pictures displayed in one second
- ***resolution***: size of each picture in pixels
- ***bitrate***: size of information in one second of video

2.2.2 The need for Compression

The previous section introduced some basic characteristics about video and to understand the need for compression a practical example can be used. Let's consider a Full-HD camera, that records video with a resolution of 1920x1080 pixels per frame at a rate of 60 frames per second. To save or transmit one minute of video, it would require to save 1920x1080x60x60 pixels and, if 8 bits

are used per pixel to represent its color, it would require approximately 7.6 Gbytes of space to save just one minute of video. Even for today's standards, it's not feasible to store one-minute videos with 7.6 Gbytes and, due to bandwidth limitations, it's also not feasible to consume 1 Gbit of bandwidth to transmit video streams. This is the biggest reason for the necessity of emerging compression techniques, to reduce space in video streams so that better quality videos can be saved and shared with lower space usage.

2.2.3 Video Coding Overview

According to [BBG15], compression is the act or process of compacting data into a smaller number of bits. The encoder converts the source data into a compressed form occupying a reduced number of bits, prior to transmission or storage, and the decoder converts the compressed form back into a representation of the original video data[Ric04a].

There are two major categories of compression, lossless and lossy compression [OM06]. In lossless compression one can decode encoded information and restore all information, that is, data decoded is the same as the one encoded, however, one cannot efficiently reduce enough size in the video bitstream without loose some information. That's why the most efficient video codecs are lossy and based on the principle that some information is redundant to the human eye/hear, there's no much harm in removing it.

A codec represents video sequence as a model efficiently represented using only a few bits, the steps to create this model are very common across most of the popular codecs used nowadays. Prediction, transform, quantization and entropy encoding are some of the most important steps to create that model and we will look in the next subsections in detail.

2.2.3.1 Prediction Model

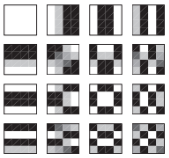
The objective of prediction is to create a representation of information. The main idea behind this process is to predict the data and subtract that prediction from the current data to create a residual. That is, based on the fact that adjacent frames have the same picture with some pixels dislocated or, in the same frame, we can represent a block of information in a picture based on the information of other blocks in the same picture.

Prediction can be temporal or spatial, depending on if data is predicted based on previously/-future frames or based in the current frame. The output of this process is a prediction model and a residual obtained by subtracting the predicted data from the current being encoded. Both of these outcomes are then encoded and sent to a decoder that re-creates the current frame.

Inter Prediction Inter prediction consists of using previous/future frames as a reference to predict the information in the current frame since residual energy is due to the movement of some pixels between two frames.

Inter prediction is computed by frames divided into MxN blocks called Macro Blocks and, the codec's algorithm finds the best match from the predictor (the one whose residual energy is

(a) DCT coefficients [kn:19b]



(b) DCT pattern

Figure 2.1: DCT coefficients and DCT pattern example [ES06]

smaller). When a match is found, that sample becomes a predictor and is subtracted from the current macroblock creating a residual block[GCDCM+13]. The Motion Vectors obtained from the offset of the predictor block and the current block and the residual block itself are transmitted to the next steps.

Intra Prediction The process of intra prediction is similar to inter prediction, the only difference is instead of using predictor macroblocks from other frames, intra prediction uses previously encoded macroblocks from the same frame. Normally, encoders use intra prediction for the reference frame - a frame that has all the necessary information to be independently decoded and, inter prediction for the other frames [sKLK03].

2.2.3.2 Image model

After the prediction model is computed we get residual energy that is lower than the original frame. To reduce information without reducing its perceived quality we must convert into another domain, the transform domain. This transformation can be done with Discrete Cosine Transform which operates on a block of N N samples, to create an N N block of coefficients [PKK+09].

To better explain DCT let's consider Fig.2.1a that represented the energy of a 4x4 block. Based on the pallet of patterns in Fig. 2.1b we can compute the weight of each pattern to describe the block. Perceived signal distortions are concentrated in low-frequency signals and one of the biggest advantages of DCT is the fact that the signal energy is concentrated around low frequencies which means high-frequency elements will have low levels of energy. This means that applying DCT to blocks of NxN pixels, low energy elements will be concentrated in the upper left corner of the block and high frequencies in the right down corner.

If mathematical distortion is applied to high-level frequencies it will not be directly perceived in the image and thus, quantization is performed by codifying with a few bits high frequencies and more bits to lower frequencies. This process is performed by dividing the DCT coefficients matrix by a well-known matrix that gives more importance to the left high corner and lowers importance to the right down corner. The bigger the quantization parameter the bigger the compression and the loss of information.

2.2.3.3 Entropy Encoder

The result of DCT transformation is a block with only a few non-zero coefficients but the encoder still has to find a technique to efficiently encode all the information.

Once the DCT coefficients are reordered the next strategy is to use Variable-Length Coding (VLC), codewords that represent symbols, the bigger the frequency of the symbols the smaller the VLC codeword, that's why the encoder needs to reorder the DCT coefficients. One very popular technique used to assign a VLC codeword to a symbol is called Huffman coding.

Now that the encoder successfully encoded a macroblock into a bitstream it must repeat the process to all the other macroblocks in the frame. Once this process is done, the encoder must save the information about the motion vectors, quantization parameter, macroblock energy, and symbol map table so the decoder so can successfully reconstruct the frame.

2.2.4 The Hybrid DPCM/DCT Video CODEC Model

The most popular video codecs since the 1990s are based on a generic model of a video codec that starts by performing motion estimation and compensation and then transforms and entropy encoding.

This model is especially important for encoders because it describes how to perform motion estimation and compensation. As can be seen in Fig.2.2 motion estimation of frame Fn is performed by comparing it to a reference frame. However, since we downgrade the current picture information, that motion vector computed is no longer correct, so one most reconstruct the current frame from a decoder's perspective to compensate for the loss of information [Wan10]. This is why encoding is a heavy operation, besides having to calculate the motion vectors it also has to reconstruct the image from a decoder's perspective to correct the previously computed motion vectors.

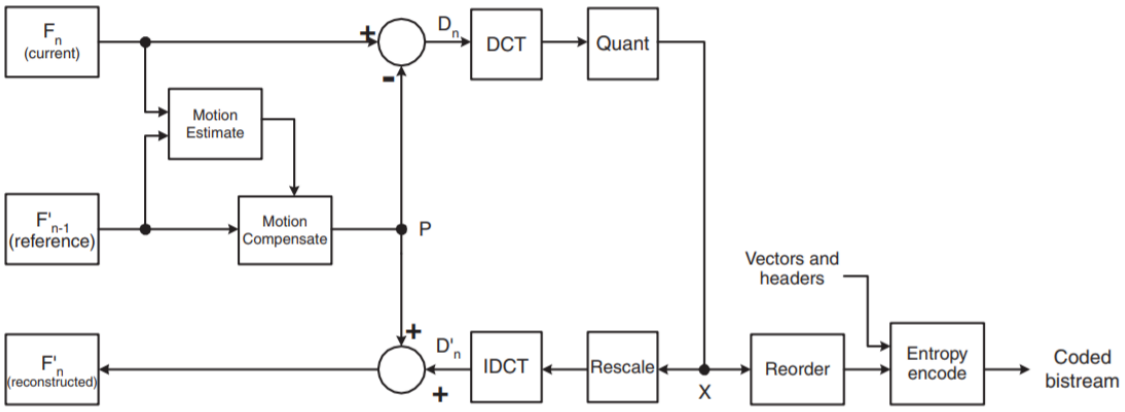


Figure 2.2: The hybrid DPCM/DCT video CODEC model [Ric04b]

2.2.5 Compression Quality Measurement

The best way to evaluate video quality is by doing it subjectively since no metric can describe it better than the human eye however, it's not convenient to do so. Several objective metrics were proposed to automatically evaluate quality.

2.2.5.1 Peak Signal-to-Noise Ratio (PSNR)

PSNR is used to evaluate the quality of a reconstructed picture compressed by a lossy encoder [AS15][LM08]. It's measured on a logarithmic scale and depends on the mean squared error (MSE) between the original picture and the reconstructed one. The main disadvantage of this metric is since it requires the original image which is rarely possible.

2.3 Multimedia Distribution Protocols

In the final of the last century, computers became powerful enough to display video streams. However, video services were transported in non-stream ways such as CD-ROMS or by downloading all the information from a remote server. With the increase of bandwidth and encoding capabilities, video streaming channels became more attractive and feasible to implement. The need for a video on demand and video streaming, where users can't wait to download the entire information to play the service, pushed the need for the implementation of several protocols that could offer the best video streaming quality for each client's capabilities. In the next subsections, we will look at some of the most popular protocols used nowadays (HLS, DASH, MMT) and the base of all those protocols (MPEG-TS). This research will allow us to better understand how data is distributed inside an IPTV system and how the new adaptive streaming protocols work. Based on the research one can evaluate whether the system could take advantage of such protocols to be implemented in the IPTV solution.

2.3.1 MPEG transport stream

To transport media streams, the MPEG-TS protocol was created to specify how information should be segmented. One of the main objectives of MPEG-TS was to transport information in links where the absence of transmission error is not guaranteed, this idea heavily influenced the design of this protocol [ZSAY00].

MPEG-TS is just a standard that specifies how information is segmented, no specification is given about how information should be encoded or multiplexed, the server can choose any codec and any multiplex strategy desired to publish their services. To design a transport stream suitable for error-prone links, small packets are used with many features from the data link layer such as packet identification, synchronization, timing, multiplexing, and the actual video information. The presence of the data link layer makes this standard perfect for raw media, however, introduces some redundancy and duplication of information when used in networks with that functionalities built-in such as IP but, it is widely used in the market and has been adopted as the standard

encoding and delivery of digital service, broadcast and as part as many other protocols as we will see next.

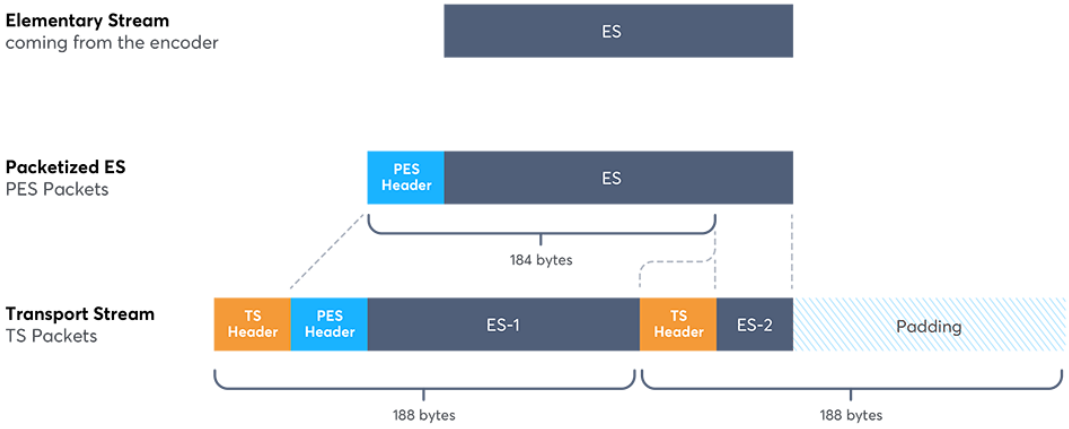


Figure 2.3: MPEG-TS Packet Struture [Tra19]

As is illustrated in Fig. 2.3 an MPEG-TS segment is composed of 188 bytes length packets of time-multiplexed information from different streams. Each packet has a header of 4 bytes containing the information already mentioned and only 184 bytes of payload, very well suited for error-prune channels. One of the key fields for this protocol is the Program Clock Reference (PCR), which can be used to help synchronization of the playback at the receiving end, very useful to synchronize different services.

Another key of MPEG-TS is the ability to multiplex several services in a single channel, using the PID value from the header of each packet it's possible to distinguish between the different services. This means that it's possible to use a PID value to identify a video signal, another to identify the sound and another one to identify the video/audio tracks inside each video stream, i.e, several services such tv channels or Video on Demand have different audio tracks with different languages.

2.3.2 HTTP Live Streaming

A few years ago the most popular service to share video on demand or live streaming was Adobe Flash Media Streaming Server, a proprietary service belonging to Adobe. However, due to the high cost and bad implementations on some platforms causing the battery to drain faster, Apple decided to not use it. This event marked the popularization of HTTP Live Streaming (HLS) protocol, created by Apple to share video-on-demand and live streaming between devices using an ordinary Web server.

One of the big features of HLS is the adaptive bitrate streams [JOY+13]. With the combination of a server and client software, HLS can detect the client's bandwidth and adapt the video stream parameters (bitrate and resolution) to better suit the client without any manual configuration. HLS,

has three components: a server component, responsible for encoding and data encapsulation; a distribution component, responsible for client management and data delivery; and a client component, responsible for selecting the right media component, download it and assemble it to reproduce.

HLS encodes a video stream into a MPEG-TS bitstream and that stream is then broken into small pieces, for example, into two-second videos. Each piece, called a segment, is indexed, stored and published in the webserver. Whenever the client wants to reproduce a given video stream, it merely has to request all indexed segments from the webserver.

2.3.3 MPEG-DASH

MPEG's Dynamic Adaptive Streaming over HTTP (DASH) is similar to HLS, it was developed by Google in response to the lack of a technology standard towards adaptive streaming. DASH is, since 2012, an ISO standard for adaptive streaming over HTTP that replaced existing proprietary technologies like Microsoft Smooth Streaming, Adobe Dynamic Streaming and is adopted by several content providers.

Like HLS, DASH segments video streams and stores them in a web server accordingly to several profiles [GH15]. Each video is represented by an XML based file called Media Presentation Description (MDP) containing all the information about the segments and profiles of the video. Whenever a client wishes to reproduce a service, he simply pulls the MDP file from the server, parses it and then, has all the segments and all the profiles to choose from to reproduce the video [SP11].

The MDP file is structured in a way that optimizes seek operations. Each video is divided into multiple periods, with information about starting time and duration.

2.3.4 MPEG Media Transport

MPEG Media Transport protocol came after the recent requirements in multimedia due to the increase of multimedia content. As described before, most content distribution protocols rely on MPEG-TS which provides efficient mechanisms to error-prune channels and services multiplexing.

One of the key features from this standard, to offer good performance in error-prune channels, is the fact that data is packetized into small packets and assigned an identifier. Because of the continuity of the sequence number, it's not possible to have a dynamic insertion of data, such as the ability to dynamically select the audio track, because once packets are stored, they can't be modified since, any alteration would disrupt the packet sequence which makes MPEG-TS not able to support some of the emerging features in content distribution.

Besides this limitation, the use of multiple services from different sources such as storage and cache would require a unique identifier linked to the package source, however, MPEG-TS identifiers belong in the context of its source, that is, different package sources have their context of identifiers and it's not possible to guarantee that different packets from different sources should have different identifiers, as it is not possible to identify two same packages stored in a nearby

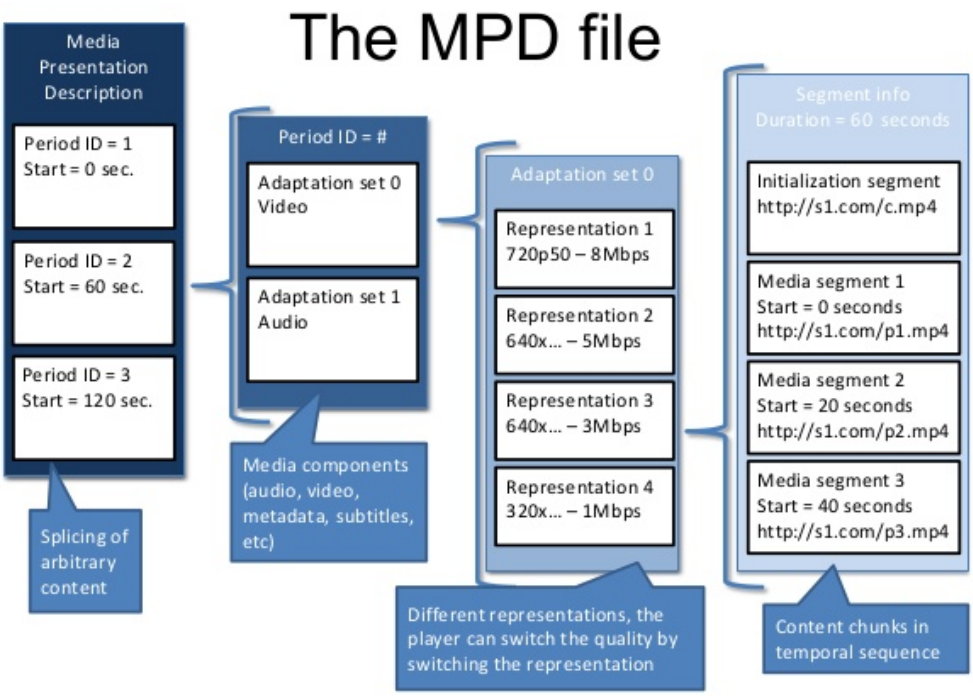


Figure 2.4: Media Presentation Description file structure [Men19]

cache system or a web server because MPEG-TS does not guarantee the same identifiers for the same package in different locations.

All these limitations from MPEG-TS and many other protocols that rely on MPEG-TS gave MPEG the initiative to create a new standard called MPEG Media Transport (MMT) [Lim13].

MMT has three functional areas to efficiently distribute content over a packet-based network:

- **Encapsulation**, which specifies the logical model and its associated encapsulation format, for timed and non-timed media content consumption.
- **Delivery**, which specifies the application layer protocol to packetize the information on the network and the payload format.
- **Signaling**, special messages describing the payload format because MMT can be dynamically changed.

2.4 Transcoding in IPTV systems

Video encoding follows complex procedures to encode data as efficiently as possible. From the research performed in how that is encoded, it is obvious to conclude that no processing can be performed without decoding at least some steps of the process. The work developed in [WZT07] had similar objectives to ours, to change the media stream characteristics to better suit the consumer requirements.

The project was developed in China and, at that time, the most popular codec was MPEG2. To reduce bandwidth, the authors proposed a Transcoder implementation which could change the video streams codec to AVS. This codec, developed in china, has better compression efficiency than MPEG2 and lower implementation complexity than H.264. Two main approaches were proposed to implement the transcoder solution, the first proposal consisted of fully decoding the video stream and fully re-encode it and, the second solution consists of taking advantage of the information in the compressed stream from the input code. In either case, the question was about transcoding complexity optimization.

Based on this solution, it was possible not only to transcode live digital TV but also to insert media content from different sources and protocols into the IPTV system. Although the author was only interested in changing the video codec, this thesis' work intends to further exploit different stream characteristics such as bitrate, frame/sample rate, and frame resolution.

2.5 Summary

This chapter started by illustrating the possibilities of an IPTV solution and, from this introduction, two key services were analyzed, Broadcast of Digital TV and VoD. The broadcast of digital TV is a one-way transmission of content from the source to multiple receivers and the end-user has no control whatsoever about the content. Many limitations can arise from the situation, one of them is the end-user not supporting the content format therefore, it becomes necessary to process the content before being sent to the consumer.

However, the media content is encoded and that process is not simple. From the studies conducted, each codec has its own logic to encode a video stream and, although the logic is very similar, it's hard to change it while data is encoded.

The distribution protocols were also analyzed and, although the adaptive streaming solutions could mitigate some of the problems referred, it would require more bandwidth, more processing to create different profiles and a server for storage. However, with these solutions, the system could process Over The Top (OTT) media content and broadcast it as digital TV in the IP network. Currently, Nonius IPTV solution uses MPEG2TS content distribution protocol and that will be this thesis main focus.

Chapter 3

Transcoding

Transcoding was developed in the early days with the primary objective of bit rate reduction to meet the available channels' capacity, however, due to the rapid growth of multimedia consumption, several video codecs were created with different capabilities and syntax. With the growing number of available video codecs not all devices can support all of them which leads to exclusion to a certain service, if a device can't recognize a video format and decode it, it can't consume the service associated to it. Due to this problem, transcoding techniques are used nowadays not only to reduce the bit rate but also to change the media stream format.

Transcoding consists of changing a given video format characteristics of a bitstream to another [LBC06], this is particularly useful to change video formats or perform space and temporal variations. In the previous sections we described how HLS or DASH create different profiles with different bitrates and resolutions, given a video bitstream, transcoders can be used to create those profiles.

Transcoding is normally implemented in hardware but, it can also be performed in software (CPU) with some heuristics. By taking into account previous encoded information, in some particular operations, it is possible to reuse that encoded input information and produce, with small changes, and output an encoded media stream.

3.1 Challenges

Video transcoding has several challenges associated with it and, from them, two stands out that must be taken into account. The first big challenge is distortion, associated with the picture quality given a bitrate. The second one is complexity in means of time/computation and memory. The picture quality is correlated to space, the bigger the picture quality the bigger the necessary space to store it. Complexity, on the other hand, is associated with the codec quality, codecs are normally improved to reduce compression size and not to be more computational inexpensive.

3.1.1 Distortion

The most efficient encoders are lossy, that is, compression is efficient because unimportant information is discarded, once a video stream is encoded by a lossy encoder some information is lost forever. Transcoding includes encoding, a given device performing this process receives an encoded video stream, decodes that same stream and reencodes it again, which results in more information losses which in turn will contribute to a greater distortion in the image. A balance must be reached between size and quality because transcoding a video sequence that has degraded input video quality will further reduce its quality.

3.1.2 Complexity

Video transcoding is a heavy operation, it consists of decoding a video stream and completely reencode that stream into another format. To overcome this problem, transcoding is usually implemented in dedicated but expensive hardware. For live-streaming applications, transcoding must be performed as fast as possible which requires a massive amount of computation otherwise, live-streaming will not be possible.

3.2 Architectures

There three main architectures proposed for video transcoding. The simplest and most straightforward way is cascaded decoder encoder transcoder. Cascaded transcoders consist of fully decode a service and fully reencode it to meet some target specifications. However, by reusing previously encoded information, the amount of computation and complexity can be saved while maintaining acceptable quality. Two architectures that reuse previously encoded information to reduce computation are Open-Loop and Closed-Loop Transcoders.

3.2.1 Open-Loop

Open-Loop is also a relatively simple architecture [VCH03]. In this system, only the quantized coefficients (DCT coefficients of the residual for a macroblock) are inverse quantized and then quantized to satisfy the output bit rate. The media stream is variable-length decoded (VLD), to extract the DCT coefficients, re-quantized and variable-length coded (VLC) again. The main advantage of those systems is that frame memory is not required because only quantization is changed and that can be performed on-the-fly.

It's not possible to reduce the bit rate at low complexity without sacrificing quality and open-loop systems are not the exception. Since the prediction vectors must be computed from the reconstructed frame with the quantization coefficients to be accurate, changing only these coefficients will invalid some motion vectors. This phenomenon is called *drift* [LW09] and is caused by a mismatch between the reference frame used by the encoder and the reference frame used by the decoder.

3.2.2 Closed-Loop

This architecture aims to eliminate the drift problem by applying a simpler cascaded decoder-encoder transcoder. To perform it, this transcoder architecture also reconstructs the frame [VLDCVW+11], the main difference, though, is the fact that cascaded architecture transcoders perform reconstruction in a spatial domain, thus requiring one DCT and two IDCT loops, and closed-loop architectures, however, only require one loop for each DCT and another for IDCT. Some inaccuracy is introduced but in most cases is imperceptible to the human eye.

3.3 Categories

There are two main categories of transcoders, homogeneous and heterogeneous, both of them can perform bit rate variations and spatial and temporal resolution. While homogeneous maintains the same video codec format and only changes spatial and temporal resolutions, heterogeneous transcoding changes spatial and temporal resolutions from one codec format to another.

Heterogeneous are easier to implement in cascaded transcoder architectures since little information can be reused from the encoded input stream due to the fact that different codecs use different processes to store the information and sometimes (especially outside a family of codecs such as MPEG) is harder to convert and process the information than to full decode and full encode.

3.4 OpenMax

With the growing demand from consumers to improve media content playback in several devices, a new class of products was created to accomplish the high-performance processing and high data throughput capabilities [kn:19a]. Examples include:

- General purpose processors with specific multimedia extensions
- Low-level hardware accelerators
- Multiple processor architectures including DSPs
- Dedicated hardware video decoders

The problem with all the possible different architectures is to develop efficient code. Since most of the code must be written in assembly, it means that software must be re-written and optimized for each new platform that it is ported to. As a consequence, it becomes harder and more expensive to introduce new products.

OpenMAX is a set of open Application Programming Interfaces (APIs) for multimedia applications with the goal of reducing costs to port software between architectures. OpenMAX APIs are common tasks independent of the architecture and thus, to introduce a new architecture, the vendor only has to focus on the specific logic of the chipset.

This thesis uses an Android Set-up Box that follows the OpenMAX standard which means, there are already abstractions to process media content and, the developed solution can work in any device that follows the OpenMAX standard.

3.5 Decoding and Encoding Technologies

There are several technologies in the market to perform decoding and encoding operations of media content and several new trends have to arise to provide transcoder resources as a cloud-service in efficient manners due to low-performance devices of the client [KCLR15]. However, this thesis work is not based on cloud computing, it is based on taking advantage of OpenMAX and multimedia dedicated hardware (DSP) to perform decoding and encoding operations. Said that and by considering this thesis' work platform, one can differentiate two main technologies namely, FFmpeg and Android MediaCodec.

3.5.1 FFmpeg

There are in the market several free software options to perform transcoding operations, FFmpeg is one of the popular options. In order to implement a transcoder, the system needs a Linux distribution instead of Android. The system could simply take advantage of the libraries offered by FFmpeg which are capable of demultiplexing a great variety of containers, at least all desired containers required by our system. The integration between the demultiplexer and the decoder is easy, and FFmpeg offers a greater list of codecs than Android. Encoding data and multiplex it is straight forward, the number of outputs that FFmpeg supports is far greater than the ones required by the system.

One big advantage of this system platform is the DSP chip described earlier that allows the system to perform decoding and encoding operations in hardware. FFmpeg can also take advantage of this chip because it relies on Linux V4L2 library however, most of the platforms that are intended to use do not officially support Linux and so, the library that FFmpeg depends on, is not implemented. Each device requires a driver so the library can be used and, in order to fully implement a hardware decoder/encoder capabilities, we would have to implement that driver. Implementing a driver requires deep knowledge of the chipset and that knowledge rarely is available to the public, besides, our system would not be portable since, for each new platform or chipset, we would have to create a new Linux kernel version with the implementation of the intended driver.

3.5.2 Android Media Codec

Android implements OpenMAX, a set of low-level media libraries written in C, the main principle is abstraction and Android, by taking advantage of this API, it has a set of open Libraries to perform operations related to media manipulation. One abstraction of those libraries is MediaCodec, a high-level framework to handle, besides many other important features, decoding and encoding capabilities.

Media Codec, fig 3.1, can be used to access low-level media codecs, it offers its users the possibility to configure a codec to decode/encode media streams. The user simply has to correctly configure the codec, write each sample in an input buffer and read the decoded/encoded data in the output buffer. Since it's the responsibility of the vendor to specify which codec is available in the platform, it varies by chipset but, most of the popular codecs are available.

However, the media libraries available in Android to demultiplex and multiplex are very limited, it can only extract from certain containers, none of which include MPEG2TS, this project's main target to perform live-transcoding in an IPTV system. In order to overcome this limitation, one would have to implement a demultiplexer and a multiplexer that could work for the most popular codecs and this solution is very difficult since it would require a lot of effort to add and maintain new codecs.

Furthermore, the media codec capabilities are limited, the encoder for example only takes an input raw media sample and encodes it. If the system is required to change the sample rate or frame size, it has to provide its own implementation of those tasks before encoding the data. FFmpeg, in comparison, already has the necessary procedures to process the samples.

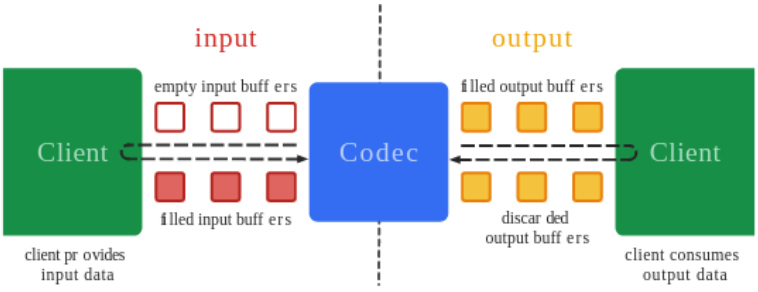


Figure 3.1: Multiplexing Components Overview [Exo20]

3.6 Related Work

Tremendously work has been developed around transcoding between them, we will cover two pieces of research in this section that resume two possible different approaches.

The first project [YLB16], a cascaded transcoder was implemented using a Raspberry Pi Board with the main objective similar to ours: build a low-cost transcoder. Raspberry Pi is well-known boards for their high ratio between performance/capabilities and cost. In that project, the author used the board at the end of communications between the data center and the end-user to reduce bitrate, by placing the transcoder at the wireless edge, such as home WiFi Access Points (APs). By doing so, the author reached better performance than most bitrate adaptive streaming protocols described in the last chapter. The only drawback of this solution is its scalability since the transcoder is implemented at the wireless edge, one instance of the transcoder must be initiated for each device, which means one can have two or more devices consuming the same service but

running parallel transcoding operations. In our project, we intend to have one transcoder per service and achieve better quality per cost since, Raspberry Pi boards are not focused in multimedia as much as the hardware used in this project for example, in the Raspberry Pi project the author mentioned he could only perform one instance of HD transcoding but, from the tests performed so far we know our hardware can perform FULL HD transcoding or even 4K transcoding. It is worth mention that the Raspberry Pi also follows the OpenMAX standard.

The second [ZXW18] project is related to software implementations of a transcoder. However, the presented solution is unrealistic for the current market requirements. For example, the used board is based on ARM architecture, similar to ours, however, the overall price per service in the presented solution is too expensive to compete against the current market approaches. Still, the author took full advantage of the board by using CPU and GPU capabilities but, in the end, this solution is too complex and not feasible as a primary target in this thesis.

3.7 Summary and Conclusions

Video transcoding is the process of changing the proprieties of a video stream encoding. It's a heavy computational operation mostly performed by dedicated hardware and can be implemented in three different architectures: Cascaded decoder-encoder, Open Loop and Closed Loop. Cascaded decoder-encoder architecture fully decodes the video stream and reencodes it into the same video stream with the desired characteristics. Open Loop architecture, in turn, simply extracts the quantization coefficients to re-quantized them to obey the new bit rate requirements. It's the simplest architecture but introduces drift due to the mismatch of the original picture and the one reconstructed. Finally, Closed Loop architecture is similar to cascaded decoder-encoder architecture due to it also reconstructing the frame to perform motion compensation but, by not doing it in the spatial domain is more efficient and the introduced errors are imperceptible.

In this project, we are using a cheap device with small computation power. However, since our device has dedicated hardware to perform decoding/encoding capabilities and takes advantage of the OpenMAX Standard, one can easily implement a cascaded decoder-encoder transcoder. Since all the operations are performed in hardware, CPU and GPU resources are available to use. In other related works, transcoders implemented in ARM architecture devices didn't go far beyond proofs of concept since the video streams used to test them are not practical for the current market requirements. Since our device supports OpenCL one could apply the related work to obtain better results by optimizing the operations to work in parallel between the CPU and GPU however, the high complexity of the required job would not be feasible if one takes into account all the necessary operations to create a transcoder.

Chapter 4

Specification of the proposed solution

To better understand the context and motivation behind this thesis' work, it is relevant to understand the solution in which this investigation is part of, the Nonius IPTV Solution. This is important because the work performed will be part of the solution. Once the Nonius solution is overviewed, the thesis work will be explained and also its challenges. The chapter will continue to further detail the problem and finally, the modular architecture of the solution and its modules will be presented.

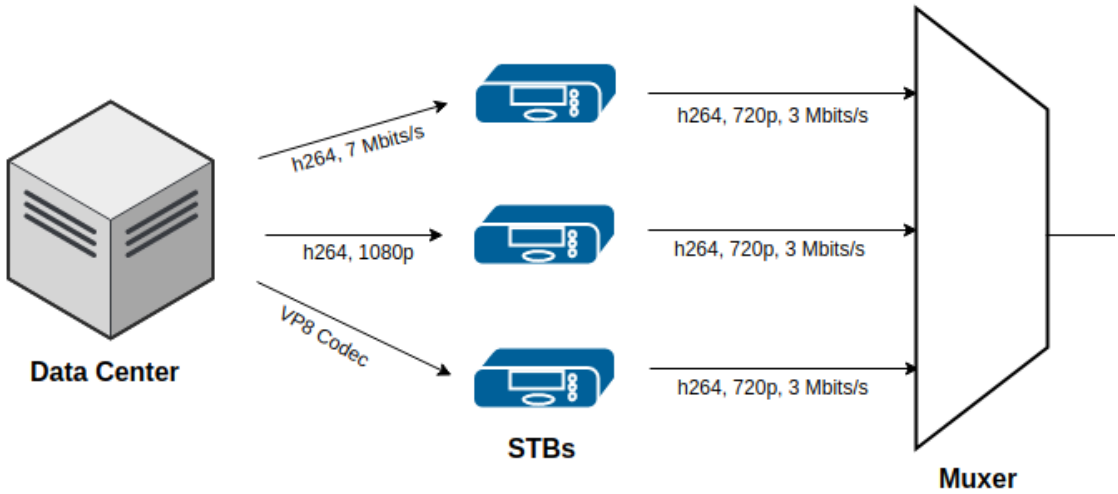


Figure 4.1: Project Overview

4.1 Nonius IPTV Solution (Solution Scenario Background)

4.1.1 Overview

Nonius is a tech company that offers technology to hospitality operators to enhance their guests' experience. The range of technology offered is large and goes from network management to IPTV or mobile solutions. All of those technologies are decoupled from one another which means a hospitality operator can have only one Nonius product integrated with other products outside Nonius.

One of the most successful Nonius' product is the IPTV system, Figure 4.2. To understand how Nonius IPTV solution is possible it's important to refer to the difference between a hospitality Television and a consumer-grade Television. Hospitality TVs differ from consumer-grade TVs due to their remote management system which allows integrators, in this case, Nonius, to create interactive pages on the TV and remotely control it.

Looking at Nonius IPTV solution architecture there are at least three components: Backend Server, PMS and IPTV Channels Streamer (besides the Hotel TVs).

- The **Backend Server** manages the content displayed on the TV, the communication between the hospitality operator and the guest and, the communication with the PMS system. Nonius IPTV solution allows the hotel operators and the guest to do much more than watching media content. In Figure 4.2 there's a representation of the TV solution, where the guest can not only watch media content but also interact with other services such as networking, weather or even request some service from the room service. The available services displayed on the TV and their disposition are configured by a hotel operator in the Backend Server.

- **PMS** system, Property Management Systems, allows hotels to enhance their customer experience and efficiency. At its core, a PMS, handles booking, check-in/check-out, room rates, and billing. Nonius integrates the hotel PMS with its IPTV solution to perform billing operations when the guest orders something from the TV or other services related to check-in/out and ratings.

- The **IPTV Channels Streamer** can be composed of more than one device but they all have the same objective, broadcast media content in a multicast network. As stated in chapter 3 the corporate environments can create local multicast networks to distribute content and that content can originate from different sources. One of those sources can be a local content provider inside the hotel operator or a national operator that sends its media content through satellite dishes, terrestrial antenna, or cable and Ethernet.

4.1.2 Nonius IPTV media content sources

As stated before, the IPTV channels streamer can be composed of different sources. Currently, Nonius offers five solutions:

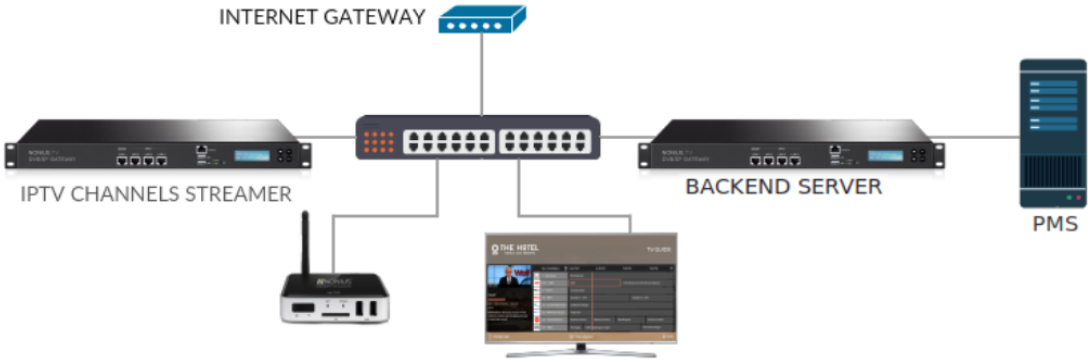


Figure 4.2: Nonius solution for media content distribution

- **Media Server**, able to store MPEG2TS media content and broadcast in the local multicast network. This device is ideal for the hotel operators to broadcast local media files such as signage, video on demand or music.

- **DVB/IP Gateway**, allows for the reception of live TV from satellite dishes, terrestrial antenna, or cable and ethernet. The content is them decoded and sent in the local IP network.

- **OTT/IP Gateway**, allows the reduction of the bandwidth of multiple requests for Over the Network (OTT) content by centralizing it into one channel.

- **IP/IP Transcoder**, allows changing the media content characteristics to support old hardware requirements.

- **HDMI/IP Encoder**, allows to convert HDMI video sources and make it available on an IP network.

4.1.3 IPTV Limitations

As stated in the conclusions of chapter 2, IPTV systems only offer one profile of media content to all the clients. From the research performed, by transcoding the media content one could reach more end users by changing its characteristics into something more suitable.

With the research conducted, in chapter 2, some IPTV systems used a transcoder to change the input stream codec into AVC. This transformation was used to reduce resource consumption due to the fact that fewer resources are required to decode these types of media streams than the others encoded with other codec and still maintain the same quality. However, Nonius already has a solution to transcode media content, the IP/IP Transcoder but, this solution is not cheap and, in order to reduce costs, Nonius wonder how well a cheap Android STB could perform the same operations.

Based on the research conducted in chapter 3, we concluded that the procedures included in an Android STB are the same as the ones required to transcode an IP stream. However, the procedures have their own meaning and limitations and are not targeting media transcoding.

Taking into account the necessity of transcoding media content in IPTV systems to support legacy hardware and the need to lower the costs, it made Nonius wonder if an Android STB, already offered by Nonius to content consumption, could transcode an IP media stream.

4.1.4 Challenges

Recalling again the conclusions taken in chapter 3 regarding the capabilities of the platform and the necessary operations to perform transcoding in IP networks, it was concluded the necessary features the system needs to perform live transcoding:

- Read media content from a channel
- Decoded media content
- Re-encode the media content into a suitable format
- Write the media content back to the channel

From the mentioned features, the Nonius IP/IP Transcoder already supports them, the objective now is to implement those features in an Android STB. It is worth mentioning that, although the investigation of this thesis is primarily focused on replicating the actions of the Nonius IP/IP Transcoder, it can also replicate the actions of the other solutions namely, Media Server, HDMI/IP Encoder and OTT/IP Gateway.

These solutions share common logic and this thesis work could combine that logic into a centralized solution:

4.1.4.1 IP to IP Transcoder

The main challenge of IP to IP transcoder is performance and quality. In order to implement a transcoder that can work in IPTV systems there are some procedures that must be fulfilled by the system:

1. **Read data from the channel**. The first challenge of the system is to get data to be decoded. That data is live content in a multicast network and therefore, the system must be able to join a multicast group and retrieve all the packets in the network.
2. **Demultiplexer**. Demultiplexing is the process of extracting elementary media streams from a given container. As studied in chapter 2, different media streams are multiplexed into a container in order to send data in only one channel. To demultiplex data, the system must be able to understand the container structure and logic and, the codec of each stream, to extract the samples' data and extra data. Therefore, the second main challenge of the system is to

extract the elementary streams from the container, which includes supporting the MPEG2Ts transport protocol and the most popular codecs.

3. **Transcoding**. As studied in chapter 2, the media content is well encoded with a specific procedure. In chapter 3 was concluded that, in order to change the stream format, one can either process the encoded data or decode the stream and re-encode it into another format. Due to the fact that the platform has a hardware decoder/encoder, it was concluded that the best option was to implement a cascaded decoder-encoder. However, the system does not have to transcode all the streams of a media stream, some streams are already in a format supported by all the consumers. The challenge of transcoding is, therefore, able to decode and encode the media streams selected from the user into a suitable format and, discard or maintain some of the streams requested by the user.

4. **Multiplexer**. In order to send the transcoded samples to the network, the system must be able to multiplex the elementary streams output from the encoder. The challenge is, therefore, be able to multiplex the samples into MPEG2TS container and support the most popular codecs namely, AVC, HEVC, AAC, MPEG-L2, and DVB subtitles.

5. **Write data to the channel**. The last challenge is to write back the contents to the network. The system must replicate the process of reading data from a multicast network.

4.1.4.2 Media Server

To replicate a Media Server the process is simple, the system already has the media content processed in local memory so, the media stream just needs to be written to the network. To accomplish this task, the process of "read data from the channel" must be able to fulfill its task in different scenarios and thus, abstract multiple stream sources from the Multiplexer. That data will then be forward to the network.

However, another process is required: **Load Control**. Load Control is a mechanism that determines when data (media content) should be read from the channel. In the previous scenario the Load Control is not necessary because the system is transcoding a live stream from the network and thus, all the packets must be processed as fast as possible. But, to broadcast a local media file, the system must keep track of the number of packets sent the network otherwise, it will overload the network with packets that will only be consumed later.

4.1.4.3 OTT to IP Gateway

The OTT to IP solution can have many different sources. Some sources can produce live content and others simply fetch a media file from a server. The point here is that most OTT services nowadays use and adaptive profile to best serve the client's characteristics and, from the studies conducted in chapter 2 this protocol is relatively new and it is common to see the hotel providers with older TVs than the creation of such protocol. This thesis work could be used to suppress those barriers, by fetching a file or live content from the network and output it to the local network. The

challenge is to identify which is the best course of action, if the problem is only the distribution protocol not supported, the system, can simply demultiplex and re-multiplex the content. If the content format is also a barrier, the system needs also to transcode the data.

4.1.4.4 HDMI to IP Gateway

This solution is exactly the same as an IP to IP transcoder, the only difference is the input media content source. Instead of reading the packets from the network, the system must read the packets from an input HDMI port. The packets are already decoded and thus, the system can encode and multiplex them. The challenge in these scenarios is to read raw media content from an HDMI input port and pass the data directly to the encoder. There's in the market Android STBs that possess HDMI ports and normally use the same chipsets as the ones found in Smart TVs.

In summary, the solutions provided by Nonius share common logic and, by re-using the logic behind an IP to IP transcoder, one system can be reused to fulfill the requirements of each solution.

4.2 Problem Definition

This thesis work arose from the limitation previously identified and the need to lower the cost of transcoding but, maintaining the capability of decoding and encoding media content, using the most popular media codecs and containers found in an IPTV system for a higher range of end-users. Since Nonius already uses ARM platforms to consume such IPTV content, that is, these platforms can demux IPTV content and consume it (decode it) and, most of those platforms offer and h264 encoder, the origin of this thesis serves as an evaluation of such capabilities in real-world scenarios.

Our solution to use an ARM platform reduces significantly the costs per service however, this kind of hardware is not intended to perform transcoding operations, even though it can perform decoding and encoding operations. This is the first problem that arises, there are no abstractions for the specific tasks that this thesis desire because, the main focus of Android STBs, is content consumption and not content processing and distribution, especially the ones with the low cost where most of the functionalities promoted by the vendors cannot be taken granted.

Another problem that arises is that most of the STBs in the market are closed platforms, that is, the code implementation is not available to the user even though Android and Linux (the base operating system and kernel) are open-source. Our chosen hardware, Amlogic, has the code available for the common user, however, there's no documentation available which becomes difficult and time-consuming to identify the needed resources and to perform code modifications that can be successfully compiled and executed.

Besides those difficulties referred, in order to maintain the cost as low as possible, computation capabilities are limited compared to other products in the market such as smartphones. Most of the researches or experiences conducted in these platforms are only proofs of concept, with small resolution and bitrates and, in order to maintain the current market requirements, this is not acceptable.

These difficulties described are the main objective of the studies conducted in chapter 2 and 3, by taking into account previous studies, their results and the requirements necessary to transcode media content, it was possible to elaborate a feasible solution for the current platform limitations.

It is worth mention that extensibility and cross-device support are two very important aspects of this plan. This system should be able to run in any device that runs Android, without any modification. This is particularly important since, as stated before, these types of low-level APIs are sometimes device-dependent. One particular example that came across was the introduction of a special set of characters while decoding h264 streams, which only occurred in one brand and was not announced by them. Of course, this is a particular case, which can only be evaluated by testing the system across several devices but other examples such as compiling platform-dependent code, C or C++ binaries, must take into account all different processors architectures. Extensibility, the other attribute mentioned, is even more important, media technology is always evolving, today, the most popular video codec used by Portuguese media providers is h264 but we are now facing a new trend and, with devices getting more powerful, the popularity of h264 video codec will be replaced by h265. Knowing this, our project architecture must be scalable enough to accept new codecs, containers or sources without having to change it, only by adding the new specific logic.

One big advantage of this solution compared to the current Nonius Solutions is its capacity to adapt to new requirements. For example, the nonius IP to IP solutions can transcode four simultaneous video streams from Full HD resolution into SD resolution. The proposed system can only transcode one stream, however, the cost per service is lower and can be adapt to the client's need. So if the client wishes to transcode a 4K video stream into an HD stream, the Nonius Transcoding solution wouldn't be able to perform this task but, since this system proposed is intended to be supported in all Android STBs, Nonius could offer an STB capable of transcoding 4K into HD. And the same is valid for other use cases, the key point of this thesis' work is to create a hardware-independent system.

In Figure 4.3 there's an overview of the proposed solution. The system proposed will not substitute entirely the current Nonius IPTV Solution because there's still the need for a DVB/IP Gateway but, it can be used in some situations to replace current solutions. It is worth mentioning that the current expectations are to use one Android STB per service, and, if the client needs to process more than one service, it will be necessary to use more than one Android STB.

4.3 High-level approach for the proposed solution

A high-level approach to the solution is shown in Figure 4.4. This is a standalone system, and a lot of abstractions can be taken granted. Those abstractions are the input and output sources, the system only has to be able to read media content from different sources, that are well known to the system, transform the information and write into an IP multicast network. To fully overview the problem it can define it in three main modules, namely Demultiplexer, Transcoder, and Multiplexer.

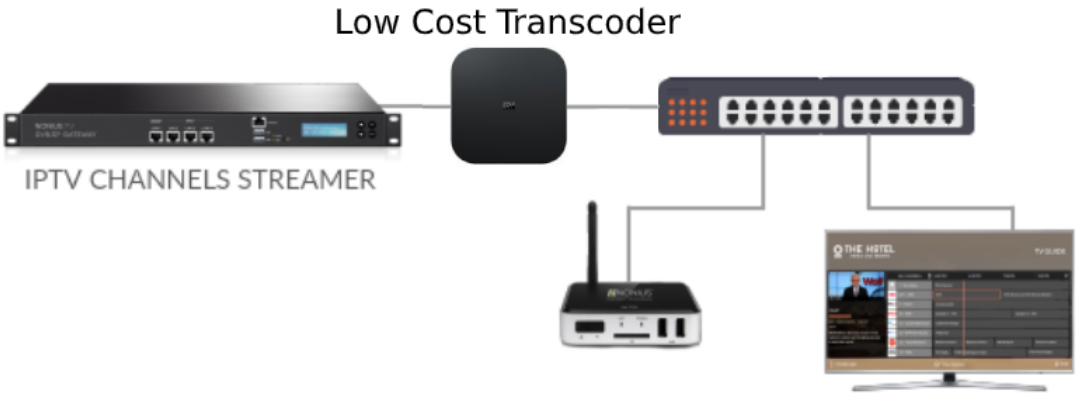


Figure 4.3: IPTV solution architecture. This solution differs from the Nonius IPTV Solution because one Android STB is placed in the network to transform one media service.

4.3.1 Demultiplexer

In order to detail the Demultiplexer Module one can divide it into three layers namely, Source, Container, Codec Reader and explain the importance of those layers:

- **Source** is an abstraction for the upstream media source. Media content can be provided from different sources, some examples can be UDP multicast networks, server files through HTTP connections or simple local files. This layer is important to abstract the creation of sockets or file descriptors to read binary data from different sources.
- As stated in the previous chapters, a media file can be composed of several media streams and, instead of using one channel per stream, data is multiplex into a **Container** and sent in only one channel. There are several containers options and each one with their limitations and purposes but, in the end, they all are used to save multiple elementary streams. Since each container has its own syntax, the system needs an abstraction that can represent each container. This abstraction will allow grouping the data by each elementary stream.
- **Codec reader** is a smaller layer inside each container. A container can support several codecs for each stream, and each codec has its own transportation syntax data and extra data describing each sample, all multiplexed inside the same channel. A Multiplexer, therefore, must be able to extract the transportation data and extra data and save it in another memory structure that represents each sample for each elementary stream. Since a container can have multiple codecs with their own syntax, the system needs another sub-layer to abstract the logic for each codec.

4.3.2 Transcoding

In order to improve extensibility, this module can be further divided into four components namely, Media Source, Decoder, Encoder and finally Media Output, represented in Figure 4.4.

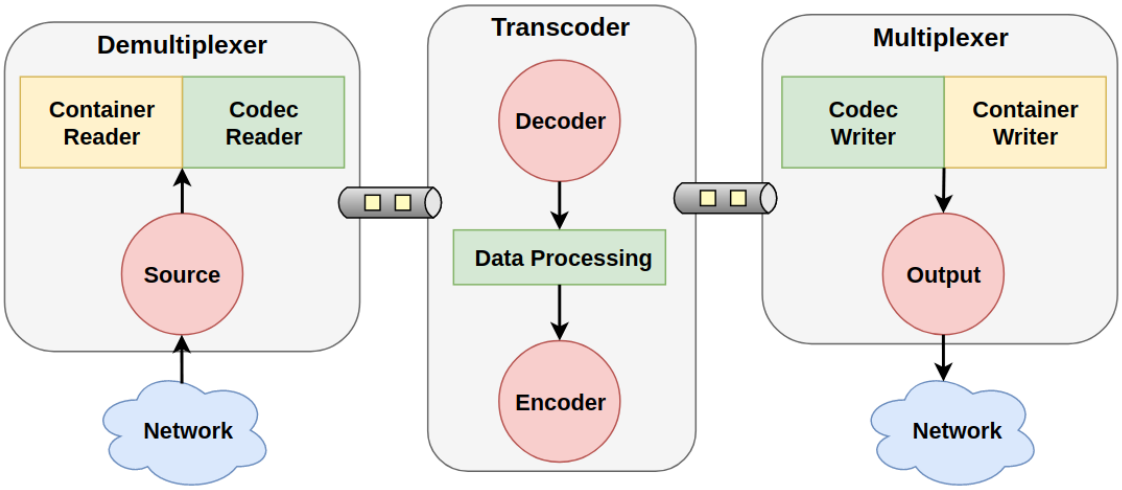


Figure 4.4: Modular Architecture

- **Media Source**, this first layer is the bridge between the Demultiplexer module and the transcoder and has two main purposes, store all the information to be transcoded and perform load control. For the first purpose, the media source layer must store all media information for each stream and its metadata. Media information is binary data previously encoded by the streamer and represents the actual data presented in a video or audio stream. Metadata, in turn, is the information that describes each sample of media information necessary to decode it (format, presentation timestamp, flags, etc). Load control, on the other hand, controls the amount of data parsed before transcoding. Since transcoding is the bottleneck of the whole operation, in some cases, we don't need to parse neither load more information into the system.

- The next layers are, respectively, **Decoder** and **Encoder**. They both work similarly. The first one, Decoder, is initially responsible to configure the codec capable of decoding the information of each type of elementary stream. Once the system has configured the codec, this layer feeds data to be decoded with its associated metadata and drains, from the codec, an uncompressed sample. The encoder performs the same operations, configures the codec, feeds uncompressed data to the codec and drains a compressed sample and its new metadata.

- Finally, **Media Output**, the bridge between the Transcoder and Multiplexer has the same responsibilities as Media Source. The first responsibility is to store all media information for each stream and its metadata (size, new presentation timestamp, new format, flags). The second responsibility is to perform once again load control. The load control in this layer differs from the last one because this time the system is not only interested in the amount of media in memory but also when the media will be consumed by the user.

4.3.3 Multiplexing

The Multiplexing process (Figure 5.8) is very similar to the Demultiplexer process, it also has the three same modules Container, Codec Writer, Output:

- **Codec Writer**, for each sample of each elementary stream, Codec Writer will write its necessary metadata.
- The **Container** will, in turn, apply its own syntax to mux each elementary stream into one media stream.
- Finally, the packets will be sent through the **Output** layer.

4.3.4 Use Case

In order to understand all modules and how data is being transformed across them, let us look at a simple use case. Consider the following scenario: a hotel operator wants to offer its guests a new channel but its video stream is encoded with the HEVC codec and some of the hotel TVs do not support this codec. To overcome this problem this system is installed in the network to change the codec. The media content is already being broadcasted in the IPTV system, the system is simply introducing a new channel with the same content but different properties.

To change the codec, the system first has to demultiplex data from an input source. The Source layer will start by opening a socket and read the binary data in the network. Once data is read, the container layer will start parsing it. Once the streams are isolated the Codec Reader can parse the information inside each stream and extract the actual stream data and the extra data.

Now that all data is well isolated, the system, can bring it to the transcoder module with the Media Source layer and, since this is live content, no-load control at the source is necessary. The next step is to Decode the information, in this case, the system will need one instance of Decoder and Encoder layers to transcode the video stream. The other streams are already supported in all Tvs so, no further processing is necessary.

The Multiplexer, in turn, will receive distinct elementary streams. For each elementary, Codec Writer layer, will add the metadata associated with each sample. Next, the container divides each elementary stream into packets. Once all data is divided into fixed-length (MPEG2TS container requirement), packets can be sent again to the network with the Output layer.

4.3.5 Solution Evaluation

In order to validate the project, we must look at the two most important metrics in a transcoder, time and quality. Since the transcoder is intended for live streaming, no matter the outcome of the implementation results, it won't be useful if the quality tests are bad. The encoder is the process that most affects it and there is some objective metrics to calculate it, in particular, the Peak signal-to-noise ratio (PSNR) that evaluates the error between the original image and the reconstructed one in decibels, the higher the value the better. However, higher PSNR values, or

other available metrics, don't mean that the human eye will perceive better picture quality, for example, two pictures can have the same PSNR value but perceived differences in quality, that's why the tests will also be subjective through observation.

4.4 Summary

During this chapter, the motivation behind this dissertation was laid out by presenting the Nonius IPTV solutions and how this thesis' work could improve those solutions by performing the same operations with reduced costs. The mechanisms behind each solution were also laid out and it was concluded that most of the procedures are common to each solution and thus, one single solution could implement them all.

The last part of the chapter specifies the modular architecture and to successfully implement the previous procedures three modules were specified:

1. **Demultiplexer** - Abstracts the extraction of media samples from different sources and containers.
2. **Transcoder** - To decode, transform and reencode the samples
3. **Multiplexer** - To abstract how the encoded samples are stored or shared.

Chapter 5

Solution Implementation

This chapter introduces the implementation details, the challenges faced and decisions took to optimize the performance and the extensibility of the system. The chapter starts by overviewing Exoplayer's demultiplexer, the integration will followup with the transcoder module and finally, the multiplexer options implemented. In order to better explain how data is handled across the system, the system will be introduced following the flow of data and focusing on some particular examples.

5.1 Technologies Chosen

In chapter 3 we introduced two solutions/technologies capable of performing transcoding or at least, some tasks that compose a Transcoder. We concluded that, due to the high complexity of implementing a codec and the solutions to reduce the cost of transcoding, the system would implement a cascaded transcoder. However, the technologies/solutions available were left open but, two main solutions were considered: Android MediaCodec and FFmpeg in a Linux distribution. Developing in a Linux distribution will require the implementation of the drivers and, developing in Android, will require the implementation of a multiplexer and demultiplexer. Since the effort to develop a multiplexer and demultiplexer is smaller, because one implementation could work across several platforms, we opted to implement our system in the Android platform.

However, implementing a demultiplexer and multiplexer requires a lot of effort, especially because it would not be feasible in the time established, to implement a capable library to extract and mux the information presented in the most popular codecs and containers. Furthermore, it would require continuous work to be compliant with new market trends. Said that we tried to look for existent open source solutions to abstract the demultiplexer and multiplexer modules and focus in the transcoding module, which is platform dependent.

Demultiplexers are normally found in media players and Exoplayer is one of the most popular media players in Android, it is fully implemented in Java and supports most of the popular

codecs and containers This media player demultiplexer suits our project, it's open-source and it was designed for the Android platform. By extracting the demultiplexer capabilities from Exoplayer we merely need to create an integration layer with our transcoder module and change some characteristics to comply with the system requirements.

Multiplexers are normally found in content distribution and there's not much use cases available in android and, the ones available, are very limited. To implement the multiplexer we focused in two approaches, FFmpeg and a custom implementation. The custom implementation is mainly concerned with MPEG2TS and only supports AVC video codec and AAC audio codec. This solution is very limited and the effort to implement new codecs and new containers is high but, the code is optimized for the Android platform and the architecture is extensible to add new codecs and containers. FFmpeg was the main approach and the one that deserved more effort, it supports most of the popular containers and codecs and, with this approach, the system could not only support video and audio transcoding but also subtitles and media streams passthrough. The main challenge was integration, since FFmpeg was not design to the Android platform and it is written in C, it requires more effort to integrate with the Transcoder module implemented in Java.

5.2 Requirements Definition

In this Subsection we will illustrate the functional and non-functional requirements of our solution. The functional requirements corresponds to the behaviors a user expects this system to have, non-functional requirements in turn, are the criteria used to evaluate globally the operation of the outline. Normally, the functional requirements are associated with the behavior for each type of user, however, our solution only has one user, a technical user that can be a real person or a server with some kind of logic to use this solution but, they both share the same privileges and thus, have the same features.

Table 4.1 lists the functional requirements of the project and which ones were successfully or partially implemented. For each requirement a priority was assign regarding its importance for the first version of the system. It is worth mention that these requirements are not the same as the ones initially defined, since none of the people involved in the project had deeply knowledge about the capacities and possibilities of this project, and thus the studies performed in chapter 2 and 3, we performed a continuous design and requirement revision before and during the implementation phase. This methodology allowed us to extract the best possible use cases for a real-world scenario project.

5.2.1 Non-Functional Requirements

There's a big list of non-functional requirements to describe our project however we can focus only on six, namely:

- **Efficiency**, this criterion characterizes the resource consumption for a given load, this is the main purpose that originated this project and the title of this thesis reflects that. The main

objective of implementing a transcoder in an ARM platform is to lower the cost given the high load that exists to perform it.

- **Usability**, it should be easy for the user to properly set up a transcoding operation without prior knowledge of the content being transcoded, the system should be intuitive to use.
- **Extensibility**, in order to keep up with the market trends in media distribution, it should be easy to insert new logic to support a new operation without needing to change the structure of the system.
- **Integrability**, in order to successfully integrate with Nonius IPTV System and for long term usage of this system, it should be easy to integrate with other products that could remotely control the system.
- **Resilience**, the system should be robust enough to take action in case of some error, it is intended for the system to perform a "best-action" effort when errors are encountered.
- **Performance**, being the main purpose of this system live-transcoding, it is important that data can be transformed as fast as live streaming is perceived by users.

5.2.2 Functional Requirements

The system function requeriements are illustrated in table 5.1

5.3 Achieving Extensibility

As mentioned before, extensibility is one of the non-functional requirements for our project, it was discussed about its importance to be able to handle new requirements without changing the architecture of the project. This section will explain how extensibility was achieved.

Java, the solution programming language, is Object-Oriented and allows the abstraction of functionalities by encapsulating them into objects and, by using hierarchies, one can further specify or generalize the behavior of a specific component. One way to generalize components is called Interfaces. Interfaces are nothing more than a document defining a set of methods that must be implemented by the objects represented by that interface and thus, one can use the Interface to refer to that object and hide the logic behind those methods declared.

By taking advantage of Java Interfaces one can easily create an interface with a few methods declared and, each instance object would have to implement the logic behind each method. The clients of that interface, only need to know about the methods declared and trust they are correctly implemented hence, increasing the abstraction between layers. If a new feature arises, a new object that implements the interface's logic could be created by simply implementing the interface and the system would know how to use it. Interfaces in Java, not only allows the creation of abstractions between modules by referring to the module as the methods offered by its Interface but also grants extensibility.

No.	Priority	Requirement Definition	Progress
US01	High	Read media from Multicast Network	Complete
US02	Medium	Read media from HTTP protocol sources	Complete
US03	Medium	Read media from local files	Complete
US04	High	Support data extraction from MPEG2TS Containers	Complete
US05	High	Support data extraction from MP4 Containers	Complete
US06	Medium	Read media from local files	Complete
US07	High	Support H264 video codec, both data extraction and decoding	Complete
US08	High	Support H265 video codec, both data extraction and decoding	Complete
US09	High	Support AAC audio codec, both data extraction and decoding	Complete
US10	High	Support MPEG audio codec, both data extraction and decoding	Complete
US11	High	Support DVD subtitles data extraction and decoding	Complete
US12	High	Support audio and video decoding simultaneous	Complete
US13	High	Media streams Passthrough (from Demuxer to Muxer)	Partial
US14	High	Discard media streams from being processed	Complete
US15	High	Video resolution transcoding	Complete
US16	High	Audio sample rate transcoding	Complete
US17	High	Video bitrate transcoding	Complete
US18	High	Video and Audio codec transcoding	Complete
US19	High	MPEG2TS Multiplexer	Complete
US20	Medium	MP4 Multiplexer	Complete
US21	High	H264 video encoding and multiplexing	Complete
US22	High	H265 video encoding and multiplexing	Complete
US23	High	AAC audio encoding and multiplexing	Complete
US24	High	Media Output in multicast networks	Complete
US25	High	Media Output to local files	Complete

Table 5.1: System Functional Requirements

5.4 Exoplayer Demultiplexer Overview

Exoplayer has many modules to handle media playback from different sources and formats, this section discusses mainly two that were extracted to implement this project demultiplexer layer.

5.4.1 Upstream Source Sub-Module

The *upstream* module is responsible to abstract the creation of connections and to read data from an upstream source. Some additional features are also implemented in this module, namely Stats and Allocation. Stats regards the data source stats, it keeps track of the number of bytes and Uri redirections. Allocation, in turn, regards to Load Control, it is an abstraction to centralize

the amount of data in memory (memory allocations to hold the media data). If all the data were dispersed, one could not easily apply a mechanism of load control.

To read data from a connection, one can specify two procedures, namely *open()* and *read()*. The *open* procedure will abstract the creation of a socket or the opening of a file. It is the responsibility of the object that implements this interface to define the logic behind opening the connection to the desired Uri. Once the connection is opened, the system can ideally start reading information and, since the Interface has a method to abstract content reading, the client of this interface, can start reading by simply specifying the output buffer and the amount of data he desires to read.

The Interface, in this case, is an abstraction of the entire module, the client does not know about the logic behind but he has the guarantee that, regarding the upstream source, the workflow will always be the same. If in the future a new requirement is added to this module, for example, a new upstream source, one must implement only the methods described in the interface and the new code changes will be isolated in this module.

Ideally, to parse information, the system would like the media information to be a continuous block of information, that can be read sequentially. However, many distribution protocols, as studied in chapter 2, don't work that way and several procedures must be implemented to transform downloaded media content into a continuous stream. This sub-module implements those procedures and allows its client to read data continuously like it is already in memory. Figure 5.1 shows the impact of this sub-module, data comes in segmented in packets from different sources and it is aggregated in one continuous segment.

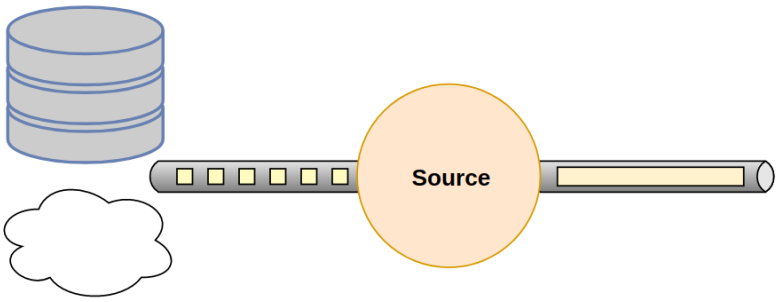


Figure 5.1: Upstream Sub-Module Data Flow Overview

5.4.2 Extractor Sub-Module

The *Extractor* sub-module is the one described previously as Container Module in chapter 4. This sub-module is responsible to parse information from a container and group it by elementary streams. This is the client of *UpStream* sub-module. As studied in chapter 2, to use only one channel with several media streams, data is multiplexed into a container that can be sent in only one channel. That container can encapsulate several programs but for now, we will only consider one program.

The Source module outputs a continuous stream of data and, that stream is composed of not only media content but also the data necessary to represent each sample and the container-specific

```
1 // Evaluates whether this extract can parse the data in ExtractorInput
2 boolean sniff(ExtractorInput input)
3 // Extractor Output buffer to write the extracted information
4 void init(ExtractorOutput output)
5 // Extracts information from Extractor Input buffer
6 int read(ExtractorInput input, PositionHolder seekPosition)
7 // Releases all kept resources
8 void release()
```

Listing 5.1: Java Interface to abstract an Extractor behaviour

data. This extractor sub-module is responsible firstly to extract the container logic data from the media content. Each container has its own logic and, for that reason, the system uses a generic interface 5.1 to represent an Extractor.

Inside each container data, there's an indication of the available media streams and which data segments correspond to each media stream. The system will use this information to group the data to each media stream. Once data is grouped, the samples are extract and its extra data necessary to decode it.

In this sub-model, several new abstractions are introduced but only two are important to understand the integration with the system. Interfaces can be used to represent a module but in this situation, they are used as a way to isolate some behavior that otherwise could became messy:

- TrackOutput 5.2, an abstraction implemented by the client of this sub-module to send the extracted samples and metadata to him. This interface abstracts the communication between this sub-module and the client.

- Extractor 5.1, abstracts the logic behind different extractors. With this abstraction the client can easily request data to be parsed or evaluate if a given extractor object can parse a give container.

Although the demultiplexer was extracted from Exoplayer some changes were performed to implement two new functionalities: media track pass-through and media track discard. To help explain, in the next sub-section, how this features were implemented it is easier to explain how the demultiplexer works in a single-use case. Let's consider an MPEG2TS stream with three tracks

```
1 // Called to write sample data to the output.
2 int sampleData(ExtractorInput input, int length);
3 // Called when a new format of an elementary stream has been identified
4 void format(Format format);
5 // Called when metadata associated with a sample has been extracted
6 void sampleMetadata(long timeUs, int flags, int size, int offset);
```

Listing 5.2: Java Interface to abstract the system storage of samples.

shown in Figure 5.2. The Source sub-module creates an abstraction of the packets read from the network and, the demultiplexer sub-module, receives a continuous stream of binary data. The data is divided in packets with length of 188 bytes, each packet is followed with a TS Header that identifies the continuity of the sample. Therefore, to extract each sample, the demultiplexer must group all packets taking into account the TS Header. However, to identify which track the sample belongs, the demultiplexer must look into the PES Header [ISO19]. This header specifies the packet identifier (PID) and the extra data associated with the sample such as size, presentation time, etc...

MPEG2TS has Program Map Tables (PMT), which describes all the tracks and their PID for a given program inside the channel. Once this table is parsed, the demultiplexer sub-module will notify the client about the available tracks. The client, in turn, for each notification, will respond back with the creation of a channel to submit all data related to the track. That channel is an instance of TrackOutput 5.2 Interface and, all the sample's data and metadata will be forward to the client using that interface.

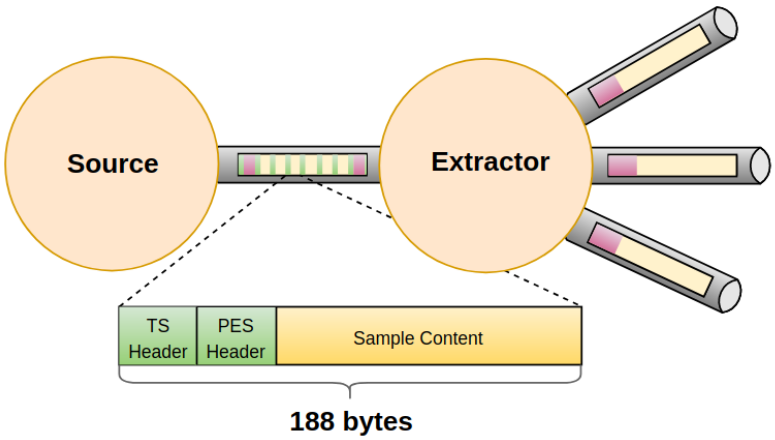


Figure 5.2: Extractor Sub-Module Data Flow Overview

5.4.3 Changes Performed

As stated before two features were implemented that required modification in the Exoplayer Demultiplexer:

1. Media Stream Discard - media content have normally more than one stream associated with it but, some streams are not relevant for the client to be part of the output media content. To avoid further processing of those streams, once the sample PID has been identified it is discarded.

2. Media Pass-Through - one of the main reasons Exoplayer's demultiplexer was integrated in the system was its optimization to the Android Platform. However, some streams were being discarded because they couldn't be decoded by MediaCodec and others lack some information. Since the stream will not be processed, the system does not have to discard

that track and, more information can be associated with the track that was previously being ignored.

5.5 Demultplixer and Transcoder Integration Layer

This layer was created to integrate the Transcoder Module with the Demultiplexer Module, as stated before, the Demultiplexer only performs actions when requested and its output, are merely samples associated with an identifier and its metadata. The integration layer handles the Demultiplexer, its main objectives are to request information to be decoded and create an abstraction layer able to group the samples and the associated data to offer a minimal interface that stores organized information.

In Lists 5.3 and 5.4, the Interfaces offered by this class are represented and in Figure 5.3 the workflow is described. Each interface represents one type of abstraction, the first interface 5.3 abstracts orders given by the client and is mostly used to perform load control actions. The *prepare()* method asks the integration layer to prepare all tracks to be decoded, the *continueLoading()* method is used to perform load control and finally, the *discardTracks* is used to easily tell the Demultiplexer to discard the processing of tracks because the client is not interested on them.

The interface described in Listing 5.4 is used to abstract the organization of data performed by this module. This interface will be used by a decoder and whenever he requests data to be decoded a new sample with its associated metadata will be passed inside DecoderInputBuffer struct. It is worth mention that most codecs, including MediaCodec, expect a complete sample to be decoded, for example, if the client wishes to decode video frames he must send a complete frame and its associated metadata to the codec, otherwise the frame will not be decoded correctly.

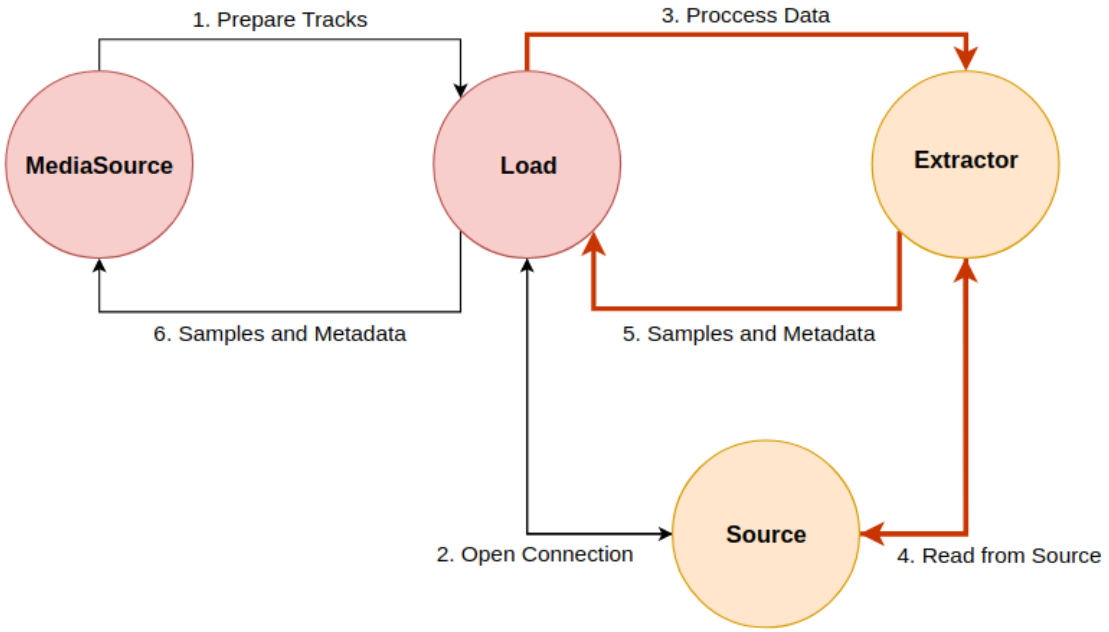


Figure 5.3: Workflow between Demultiplexer and Transcoder Module (Integration Layer)

The workflow of this layer is described in Figure 5.3, here we introduce two new sub-modules, MediaSource and Load, the sub-modules Extractor and Source are part of the Demultiplexer. The main sub-module of this layer is MediaSource, this is the communication channel with the clients that wish to use the Demultiplexer. The first question a client will ask this layer is to prepare information to be processed and, *MediaSource* will, in turn, instantiate a Loader and demand it to prepare the information about the available tracks.

A Loader is a while loop thread with only one function, demand data to be parsed and ask the client if he wants to continue parsing information. The Loader will open a connection to the data source and ask the extractors which one is able to parse the information that he collected from a source. The first extractor that can successfully parse the information (an extractor can parse the information if he can handle the container) will be responsible to handle the information, that is, at this point forward the chosen Extractor will communicate with the Source to parse information.

The Loader will then keep requesting information do be parsed and the Extractor will respond back with processed information. That response can either be the format of a new track discovered or sample data and extra data. In the case of a new Track discovered, the Loader will ask MediaSource to save the new track and create a new channel of communication with the Extractor for those samples. Once all the tracks have been identified by the Extractor, the MediaSource layer will request the user which tracks he wants to discard and, data, starts to be parsed: the Loader will keep asking the Extractor for information and the extractor will write that information in the object returned by the Looper.

Load Control is also performed by the Loader, whenever a client instantiates a new MediaSource he must define the maximum number of bytes to parse. Once that number of bytes has been reached, the Loader, will block and notify the client. To unlock the Loader the client must request him to continue loading. Using this mechanism the client can easily perform load control, our strategy was to evaluate the number of bytes hold in memory with parsed information, if that number was smaller than a certain threshold we would ask the Loader to continue loading otherwise we would let him stop.

This heuristic requires some enhancements, we did not have any troubles so far because we are transcoding live content and the load is already being controlled by the number of packets in the network. However, if we desired to broadcast a file in a remote server we would have to control the presentation timestamps otherwise, the system will overload the network and the client with packets that would take a while to be presented. This type of load control was implemented and will be described in the MediaOutput layer.

One way to evaluate the processing capabilities of the system is to look at the amount of data that is buffered in memory. If that buffering keeps growing it means that the system can't hold the processing of information, some information will eventually be lost and the client would not have a nice experience when playing back the media contents. The next chapter will look further into details with some experience results.

It is worth mentioning that the communication between this sub-module and the Demultiplexer is performed using the interfaces that describe the Demultiplexer and if the user wishes

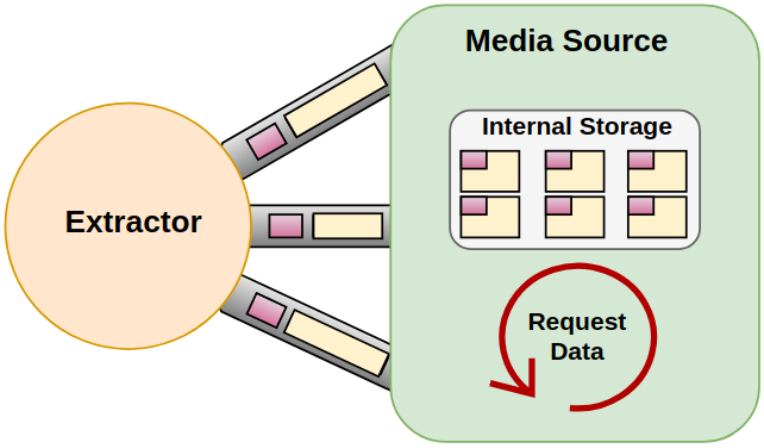


Figure 5.4: Data processing in MediaSource

to use a different Demultiplexer than the one available in the system he just needs to provide an implementation of the interfaces used by this sub-module.

In summary, the impact of this layer can be illustrated in Figure 5.4. The layer has two main functionalities, the first is **request data** to be read from the source and extracted, the second is **internal storage** of data and metadata organized by each sample. From these two functionalities, it was possible to abstract the Demultiplexer behavior for the Transcoder with a single method call 5.4.

5.6 Transcoder Module

The Transcoder Module is the heart module of the system, its main objectives are to prepare the input and output source, manage the system load and demand samples to be read, decoded, encoded and written to the output source. In the next subsections, we will look in-depth on how the system is managed and the problems faced with transcoding.

5.6.1 Client Interactions

The Transcoder module abstracts the modifications performed to media streams, these abstractions are, among others, setting up the input and output sources and transcode media streams. The

```
1 //  Prepares this media media source.
2 void prepare(Callback callback, long positionUs);
3 //  Attempts to continue loading.
4 boolean continueLoading(long positionUs);
5 //  Tells the extractor to discard the tracks
6 void discardTracks(boolean discard, int[] tracksID);
```

Listing 5.3: MediaSource Interface to abstract integration layer

```
1 //  Tries to read data from the source
2  int readData(FormatHolder formatHolder, DecoderInputBuffer buffer, boolean
       formatRequired);
```

Listing 5.4: Interface to abstract communication with the transcoder

abstraction of the transcoder is represented in Listing 5.5 and will be explained with the help of the workflow represented in Figure 5.5. We introduce here three new sub-modules namely, Renderer, Codification, and MediaOuput, the first two are just wrappers for the decoder and encoder and will be reviewed in the next subsections, MediaOutput is the Integration Layer between the Transcoder and Multiplexer.

As can be seen in Figure 5.5 whenever a client wishes to transcode media content he must first define the input and output sources. The transcoder interface offers two methods to perform those operations, setDataSource, and setOutput, these methods expect an instance of the integration layers' interfaces (Demultiplexer and Multiplexer integration layers). Our system has one implementation of each interface but if the client has its own implementation the Transcoder will use its implementation if the corresponded interfaces (5.3, 5.4) are implemented.

Once the demultiplexer and multiplexer to use are defined, the client will request the system to identify the tracks available in its media content, this method will set up the media source and media output by calling its prepare method. Once Media Source responds back and thus, all available tracks have been identified, the Transcoder forwards the information to the client and expects a response back to know which tracks will be discarded and which will be kept. The user will select the tracks he wants to transcode and specify its codification format and the tracks to be discarded. Not all tracks must be transcoded or discarded, some of them are already in a format that suits the client, those tracks will be the difference between the tracks wished to transcode and the ones wished to be discarded and will be forward from MediaSource to MediaOutput. Once all tracks are selected, the system does not need additional information and so, the transcoding process can start.

Whenever the client wishes, he can call *getStats* to get some statistics about the current progress of transcoding such as time elapsed, number of decoded/encoded, percentage of buffering data in memory, etc. To stop Transcoding, the client, can solely tell the transcoder to stop.

5.6.2 Decoding and Encoding Operation

In the last sub-section we introduced the concept of Renderer and Codification, a Renderer is nothing more than a wrapper that abstracts the logic of writing into a codec capable of decoding and Codification a wrapper to perform the same abstraction of a codec able to encode. Renderers and Codifications will be the main topic of this subsection, we will look into some detail how they were implemented.

```
1  // Sets the transcoder dataSource, should be called immediately after the
       constructor
2  boolean setDataSource(DataSource dataSource);
3  // Sets the transcoder mediaOutput , should be called before startTranscoder()
4  boolean setOutputSource(MediaOutput mediaOutput);
5  // Tells the transcoder to prepare itself
6  public boolean prepare();
7  // Starts transcoding
8  boolean startTranscoder();
9  // Sets the tracks to be transcoded, must be called before startTranscoder()
10 boolean setSelectedTracks(TrackGroup[] selectedTracks,TrackGroup[] discardedTracks,
       MediaFormat[] formats);
11 // Stops the transconder and releases all kept resources
12 boolean stopTranscoder();
13 // Requests the current transcoder stats
14 void getStats();
```

Listing 5.5: Java Interface to abstract a Transcoder

5.6.2.1 Media Codec

In order to access the low level hardware codecs, Android provides a library called MediaCodec. To start using this library a client must first get its instance, provided by MediaCodec, by specifying the sample type. So, for example, if the client wishes a codec for AVC streams he must query an instance of type "video/avc" and the return value can either be a MediaCodec instance or null if the codec is not supported. The supported formats are provided by the hardware vendor in a file called "*media_codecs.xml*".

Once the client has an instance he must specify the characteristics of the stream, for that, he must configure a MediaFormat instance and specify the characteristics required to decode/encode a give stream type. Another configuration that can be specified is the output Surface, a Surface can be compared to a Frame and it's especially important for video codecs.

Once the configuration is performed, the user can start the process of decoding/encoding samples, this process is performed iteratively through buffers:

1. The client requests MediaCodec for an input buffer and the system responds back with a buffer or null if none is valid. The buffers are natively allocated in memory by MediaCodec and he keeps the ownership of the buffer, the client is only allowed to read into the buffer.

2. The client writes the sample data into the buffer, along with the extra data associated with the sample (size, presentation timestamp, flags) and notifies MediaCodec to process the data.

3. The Sample is processed by MediaCodec

4. The user requests an output Buffer to read the processed data, MediaCodec will respond back with a buffer containing the data or with some other message if the data has not yet

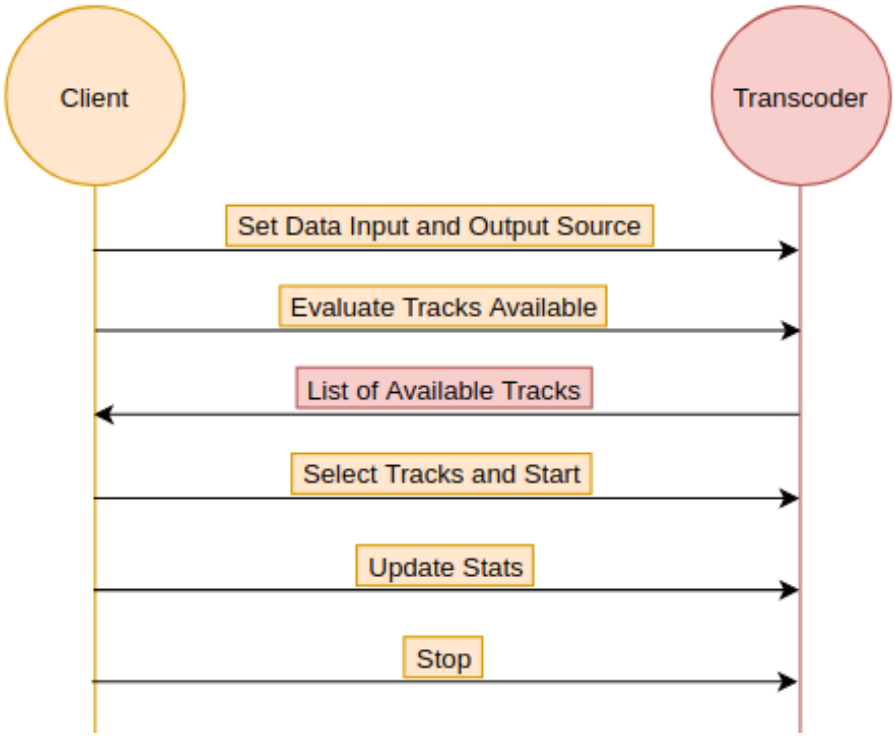


Figure 5.5: Workflow between the Transcoder Module and the Client

been processed. Again, the buffer is natively allocated and the client can only read from the buffer.

5. Once data has been read from the output buffers the client must tell MediaCodec to release the allocated data otherwise, the output buffers will fill with data until the limit is reached and at that point, MediaCodec will no longer lease more Input Buffers.

5.6.3 Media Codec for Transcoding

To transcode an elementary stream it's required two instances of MediaCodec, one to perform sample decoding and another to perform the encoding. Let's assume as example the system was requested to decode a video stream to change its bitrate: it would have to configure an instance do decode the video frames and another instance configured with a lower bitrate to encode each frame. Following the steps introduced previously, in order to decode the stream we would have to request an input buffer, write a video data frame provided by MediaSource and request an output buffer to read the decoded data (raw video data). This data will then be forward to the encoder's MediaCodec instance which would repeat the same process and forward the encoded data to MediaOutput, this process is shown in Figure 5.6.

There are two distinct ways to transfer the data between one MediaCodec instance into another instance namely, ByteBuffer to ByteBuffer or Surface to Surface. The first method is the most intuitive, we request the output buffer's data (which is a ByteBuffer) of one MediaCodec's instance (decoder) and copy the contents into the input buffer of the other instance (encoder). This

method is particularly useful when we wich to modify the contents of the stream since ByteBuffers offer direct access to pixels however, copying the contents from one buffer to another adds a big overhead.

The second method has better performance, while configuring each instance of MediaCodec, the system, can request a surface as input for a codec (the encoder) and configure the decoder to render the raw frames' data into that surface that is, the output data of the decoder will be rendered into the input of the encoder and thus, no memory copy is necessary.

Surfaces don't offer direct access to pixels but we can use OpenGL shaders to apply filters to a frame. It is worth mentioned that the process of rendering provided by MediaCodec is very limited, if the client only wishes to perform bitrate transcoding, no problems will arise, however, if the user wants to perform transcoding to change frame resolution or frame rate modification he will have to specify how the frame must be rendered and which frames will be rendered.

Our system uses Surface to Surface transcoding for video streams and uses OpenGL to perform pixel interpolation in order to change the frame resolution. Frame rate transcoding is not fully supported, we only allow the user to transcode the frame rate to half of the original. OpenGL rendering adds a big overhead while performing transcoding operations and in chapter 6 we will evaluate it. ByteBuffer to ByteBuffer transcoding is used for audio streams in this system. This process can be seen in Figure 5.7.

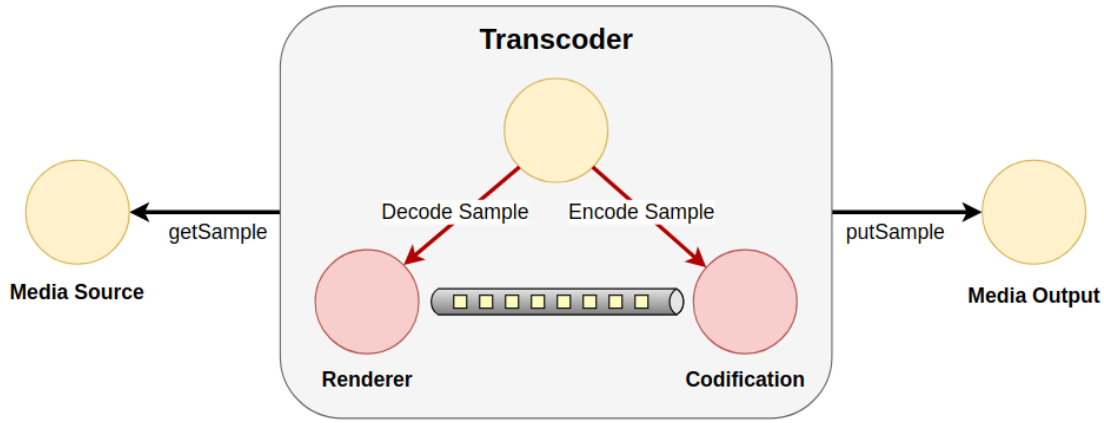


Figure 5.6: Transcoder Architecture Overview

5.6.3.1 Transcoder Module Architecture

The process described above is valid for only one stream, the architecture to replicate the same process for multiple streams is identical, each stream must create two MediaCodec instances to decode and decode data. Each instance has specific characteristics that must be configured according to the media stream's format and, so is the way data is processed. Two abstractions are used to hide those particular characteristics called Renderer and Codification, in this subsection the procedures hidden by those abstractions are specified.

There are three types of renderers and codifications that hold operations for each type of stream, audio, video and passthrough. Their architecture is abstracted in three hierarchies:

1. The first hierarchy is responsible for communication between MediaSource, it abstracts the correct identification of data to be read/write and is common to all instances.

2. The second hierarchy extends the functionalities of the last one and implements codec specific communications operations such as configuring a codec and correctly feed and drain data to the same. This hierarchy holds the logic for passthrough and for MediaCodec general operations for audio and video.

3. The third is more specific and implements all the necessary operations in compliance with the type of media streams, for example, implementing the communication between MediaCodec instances for each media type (audio or video).

In normal system usage, when the user asks the system to start transcoding, the last, responds back with a request to select the tracks, the process of selecting tracks is performed by the Renderers and Codifications. This operation consists in creating specific renderers and codifications for each of the tracks selected to transcode (video and/or audio renderers/codifications) and for the tracks selected for passthrough (called empty renderers/codifications).

To create a video Renderer it's necessary to specify the output surface, as referred earlier, video renderers, render the output frame into the encoder input surface. The system must configure the decoder with the surface provided by the encoder and, to perform this operation, the system must first configure the codification, get its input surface and finishing configuring the renderer.

For audio renderers/codifications however, the renderer output data is copied into the codification input buffer, a simple queue can hold the information provided by the renderer and the codification pools data from the queue. To reach better performance the renderer can queue the output buffer (and not its data) provided by MediaCodec and thus, only one copy of the information is necessary. However, special care must be taken in the codification to release those buffers once they are copied otherwise, this operation will eventually block all the operations.

Information passthrough does not require rendering. The system specifies empty renderers/-codifications, that use the same mechanism as audio streams to transfer data from MediaSource directly to MediaOutput without any copy, (the same data is queued and polled). The contents of this paragraph can be seen in Figure 5.7

Once the renderers and codifications are configured, the Transcoder creates a new thread to handle the decoding and encoding operations, this processed can be overviewed in Listing 5.6.

Both Renderers and Codifications implement the same two basic functions namely, *feedInputBuffer* and *drainOutputBuffer*. The first function loads data into the codec's input buffers and the second drain the available data from the codec output buffers, the source, and destiny of data for the codecs was already specified during configuration. Encoder's *drainOuputBuffer* calls will lead the Codification to fetch data from the codec and send them to Media Output, since this is the stage that requires more processing, it is necessary to make sure all data is drained from the

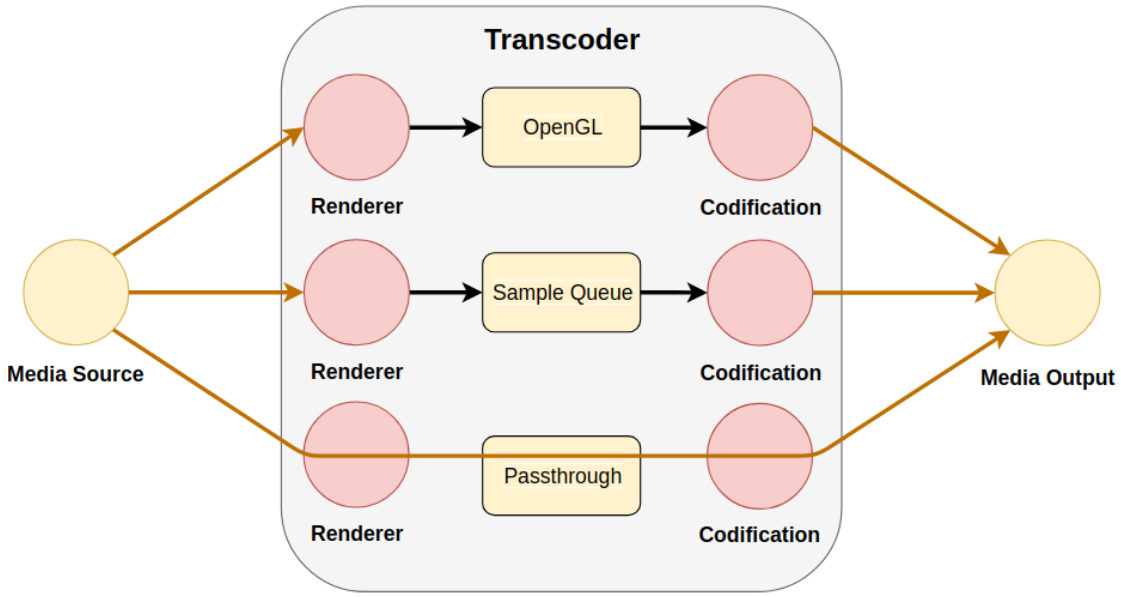


Figure 5.7: Data flow from Media Source to Media Output. This figure also describes the communication mechanisms between audio and video. Different stream types must be have similar data flow behaviour but data is not render nor codified.

encoder output buffer before feeding him with more samples to be decoded otherwise, congestion will heavily impact the system performance. In Listing 5.6 the process is described.

5.6.3.2 Rendering audio and video streams

Some special operations must be performed to render video and audio streams that are otherwise abstracted when using Android MediaExtractor :

- Only complete samples must be written to the codec
- When using Android Media Extractor, the system has the guarantee that when one sample is requested to be decoded it will return a sample to be feed into the codec's input buffer previously requested. In live transcoding, that sample may not exist yet and the system must reuse the input buffer until data is available to be written.
- Some MediaCodecs instances expect the first frame to be the format of the stream, as rule of use, the system always provides the format before start feeding information.
- For some streams, MediaCodec must be configured with the codec extra data, for example, AVC decoders expect configuration information about the "Sequence Parameter Set".
- When decoding video streams the first frame feed to the codec must be a keyframe.
- MediaCodec does not re-sample streams neither change its characteristics (only bitrate) so, if the system wants to perform those operations, they must be performed before sent to the encoder.

```
while (!canceled) {
    for (int i = 0; i < renderers.length; i++) {
        // extract all data from the encoder
        while(ret == SAMPLE_PROCESSED) {
            ret  = codifications[i].drainOutputBuffer();
        }
        // feed the encoder with one decoded sample
        codifications[i].feedInputBuffer();
        // extract a decoded sample from the decoder
        renderers[i].drainOutputBuffer();
        // feed an encoded sample to be decoded
        renderers[i].feedInputBuffer();
    }
}
```

Listing 5.6: Transcode loop to decode and encode frames

Those are the main highlights the system must check when decoding information, once they are all verified the process of rendering is as simple as asking the codec an input buffer, write the sample data along with the extra data and request the output buffers with the decoded data. In the case of video streams, the renderer must ask MediaCodec to render the information into the encoder input surface. In the case of audio streams, the renderer must queue the codec output buffer. Special care must be taken to ensure that every requested output buffer is always released otherwise the system will block.

5.6.4 Encode audio and video streams

Encoding information also as some particularities, some of them are abstracted by MediaMuxer others are just common rules to increase performance and reliability:

- Like renderers, when an input buffer is requested the system must reuse the buffer if no data is written.
- The extra data associated with some codecs will be passed when the format is discovered or when the encoded sample has the associated flag "BUFFER_FLAG_CODEC_CONFIG".
- The encoded data is only related to the sample, any extra data necessary to transport or save it must be added by the system.

The codification process is simpler than the rendering process, the data transcoding characteristics are already set and the codec just needs to encode that data. To encode data the system needs to request an input buffer, write the data to the input buffer (if encoding audio or passthrough, in video streams the data is already in the input buffer) and request an output buffer to read the encoded data. That data will be forward to MediaOuput which, in turn, must copy the data so the output buffer can be immediately released.

```
1
2  // Set some properties. Failing to specify some of these can cause the MediaCodec
3  // configure() call to throw an unhelpful exception.
4  format.setInteger(MediaFormat.KEY_COLOR_FORMAT,
5          MediaCodecInfo.CodecCapabilities.COLOR_FormatSurface);
6  // Bitrate in bits per second
7  format.setInteger(MediaFormat.KEY_BIT_RATE, 2888608);
8  // Frame expected as input
9  format.setInteger(MediaFormat.KEY_FRAME_RATE, 25);
10 // Key frames interval in sencods
11 format.setInteger(MediaFormat.KEY_I_FRAME_INTERVAL, 1);
```

Listing 5.7: Setting the encoded stream bitrate

5.6.4.1 Bitrate or Codec Transcoding

Performing Bitrate or Codec Transcoding is straight forward since it only affects the encoder's processing and there's no operation necessary to change the way frames are rendered. When creating the codec for the encoder, the system needs to create an instance of MediaFormat. This instance holds the information about the raw data that will be feed into the encoder, such as frame rate and codec, and can be used to set the output bitrate of the encoded data. In Listing 5.7 there's an example that describes how this process is performed. The example describes a video format but the same holds for the bitrate in audio formats.

5.6.4.2 Frame/Sample rate Transcoding

MediaCodec can only process frames, when the system decodes a sample and tells the encoder to encode it, the encoder will do it no matter the frame rate set in MediaFormat because, MediaCodec, does not drop samples neither performs any operation of re-sampling. It is up to the system to resample the video and audio streams. In the case of video, the easiest solution is to drop some raw frames, in audio however, the system must have an implementation of a re-sampler algorithm. The project developed only supports video frame rate and audio sample rate transcoding. For video frames, the only operation allowed is to divide the original frame rate by two, that is, every two frames one is dropped and thus, we only feed the encoder half of the initial frames. Audio re-sampling is performed using FFmpeg.

5.6.5 Resolution Transcoding

To perform resolution transcoding the renderer must be able to perform pixel interpolation, this process is not easy but since Android has a library to handle OpenGL operations, pixel interpolation can be performed in hardware and thus, accelerating the process. In Android CTS tests there's an implementation of this process that was reused by this system. Using OpenGL to render frames will heavily impact the system performance and will be shown in chapter 6 with some tests and discussion.

5.6.6 Live transcoding AVC and AAC in Android Platform - An Example

Before multiplexing the data provided by the encoder the system still has to perform some operations to that data. AAC streams are normal sent in ADTS packets, each sample, before multiplexed, must have its associated ADTS packet headers that specify the packet length, and stream characteristics such as sample rate, channel count and profile. That means once AAC sample data is read from the encoder the system must construct and pre-append an ADTS header. AVC or even HEVC is a little bit simpler, the encoder will provide two configurations, Sequence Parameter Set and Picture Parameter Set, two parameters to configure the consumer playback decoder. This information will be the same along the encoding cycle and must be pre-appended to each keyframe. Failing to specify these headers will cause the stream to not be playable.

5.7 Media Output and Multiplexer Layers

Media Output and Multiplexer are very similar to Media Source and Demultiplexer layers. Media Output is just an abstraction to control how tracks are added to multiplex and when should the multiplexer actually start, this layer was implemented to freely ask data to be multiplex.

During the process of transcoding, many events happen that must be informed to the multiplexer layer and that communication is abstracted by Media Output. The first event is to initialize and choose one Multiplexer that can handle the output container requested by the user, currently, there are three multiplexers in the system: Media Extractor from Android that can only mux mp4 containers, FFmpeg and the system own implementation for MPEG2TS. Once the multiplexer is initialized, the transcoding process starts and new tracks are muxed to be encoded. The job of Media Output is to create a Muxer Input instance to handle the Multiplexer input information and abstract stream content identification. The codifications will then start sending encoded information to muxed. This process is illustrated in Figure 5.8.

The most important aspect of Media Output, however, is to add a new form of load control. Previously was mentioned that the system had a mechanism to control the amount of media streams data in memory. It works fine when transcoding a media file or when data read operations are controlled however, it can't control the output flow of media. Media Output implementation of load control consists of buffering data until it can be sent into the network. To ensure this, Media Output takes into consideration the program clock reference (PCR) to decide when and which samples should be sent and, if the buffering gets too high, Media Ouput will propagate that information to the Transcoder module to pause the transcoding process. With this mechanism, the system not only garantees the correct flow to broadcast live content but also helps to minize unstable media transcoding delays that could otherwise propagate to the consumer.

5.7.1 Java Native Interface and FFmpeg

The first iteration of the system only supported Android MediaMuxer to multiplex information, it has very big limitations due to the lack of supported containers but a Multiplexer interface was

```
1  // Stops muxing data. Should only be called once endOfStream is encoded
2  void stop();
3  // Adds a new track to be muxed, should ideally be called before #start()
4  int addTrack(MediaFormat newFormat);
5  // Starts muxing data, should be called once all tracks are added
6  void start();
7  // Evaluates if this muxer can support this specific container
8  void sniff(int container);
9  // Writes a given stream parcel to the output
10 void writeSampleData(int trackIndex, EncoderBuffer outputBuffer);
```

Listing 5.8: Java Interface to abstract Multiplexer layer

created to abstract that module 5.8. Once the system was fully integrated with that abstraction that refers to muxers, it can support any other muxer that implements the abstraction. After some research, FFmpeg was the most promising alternative to Media Muxer with its vast support of codecs and containers however, it is a C library and the system is in Java.

Android offers a Java Native Layer that allows code running in a Java Virtual Machine to communicate with native code. In order to run native code, it must be compile into a binary file and four compilations were performed of the system to reached all Android chipset architectures.

FFmpeg is just a library, to implement the multiplexer interface the system needs two layers, one to setup and control FFmpeg, written in C, and another in Java to represent the interface and translate the calls to native calls.

The final version of the system used in tests uses only the FFmpeg library to multiplex information and broadcast it to a media output destiny.

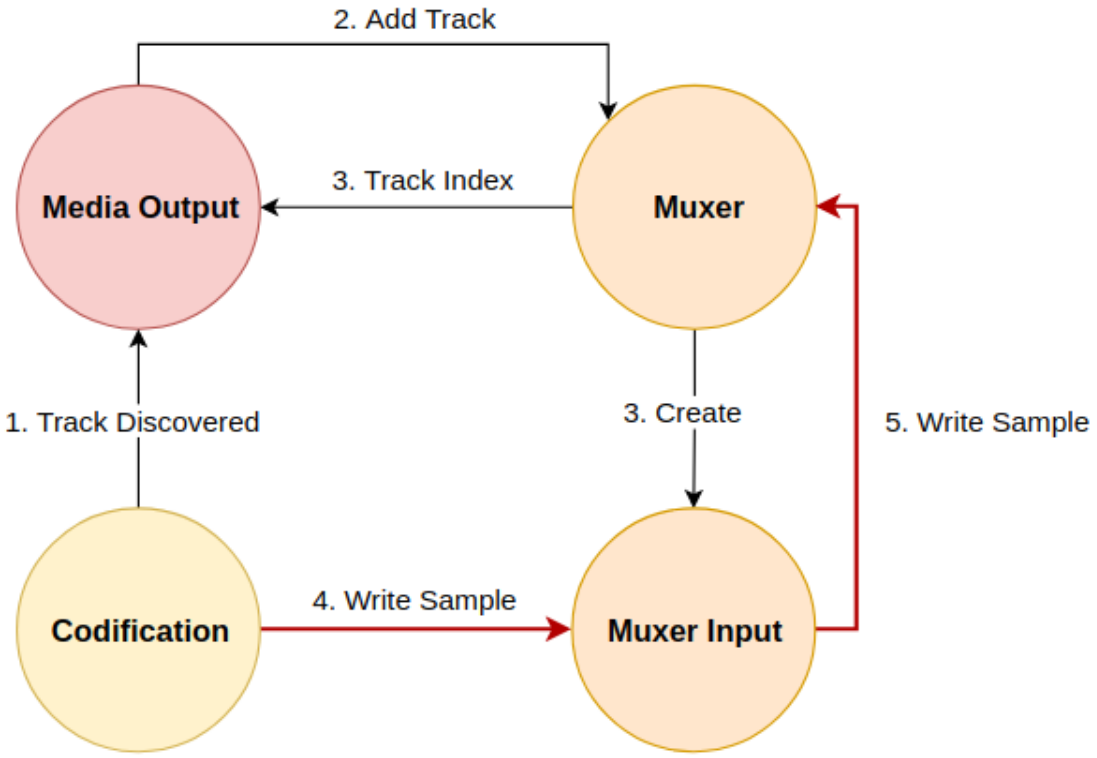


Figure 5.8: Media Ouput and Multiplexer Modules, different colors represent different modules/layers and red lines represent cyclic operations.

Chapter 6

Obtained Results and Discussion

In this Chapter, the scenario and hardware used to test the system are specified and test results are shown and discussed.

6.1 Scenario Overview and Hardware

There are two essential metrics desired to evaluate the system capabilities, quality and performance. One popular metric to evaluate the quality of a transcoding stream is PSNR, the ratio between the original data and the noise introduced by the encoder, however, it does not consistently reflect human perception, sometimes two different frames with the same PSNR values will look different for the observer. Performance, in turn, is the capacity of the transcoder to manipulate data efficiently, for live-transcoding, that capacity reflects in a smooth output stream without breaks.

Various tests were performed based on two categories, offline and live content tests. Offline tests, performed in static media content, are used to evaluate the quality of the transcoded samples. Since quality and performance will influence the results of each other, transcoding static content allows quality, of the transcoded data, to be compared across different chipsets more consistently. Some performance tests will also be performed for offline content to evaluate some overhead added by a specific operation. Tests performed in live-content have, as the main objective, evaluation of the system's global performance.

Four chipsets were used to compare the results of the system performance and quality, one chipset belongs to Qualcomm, the other to Realtek and the rest to Amlogic, the characteristics of each device can be seen in table 6.1. The tests were performed with three different streams and their video characteristics are available in table 6.2.

Device:	CPU	GPU	Decoding	Encoding	Price
Snap 820	Kryo	Adreno 530	up to 4K H265	H264 1080p H265 4K	400$
S905x3	4x A55	Mali G31	up to 4K H265	H264&H265 1080p60fps	60$
S912	8x A53	T820 MP3	up to 4K H265	H264&H265 1080p60fps	100$
RTD 1296	4x A53	T820 MP3	up to 4K H265	H264 1080p60fps	45$

Table 6.1: Chipset Capabilities Sheet

6.2 Bitrate Transcoding Tests

The first set of tests performed is bitrate transcoding of video streams. The main objective is to reduce the video quality by reducing its bitrate and evaluate the obtained results. Two different media streams were chosen, one in HD resolution and the other in Full HD. In order to transcode the media stream, the system must demultiplex each elementary stream, discard all streams except for the video stream and multiplex the video stream into MPEG2TS container. Audio will not be considered initially, most of the operations applied in audio streams will be passthrough or change the codec.

6.2.1 Bitrate Transcoding with HD Streams

In Figures 6.1 and 6.2 there's the representation of two tests performed to evaluate performance and quality. The stream used was recorded from a German channel called "3Sat HD". The video stream of this channel has an average bitrate of 12 Mbit/s with some spikes of 30 Mbit/s, the number of frames per second is 50 and the duration is about 74s.

In terms of performance, the system can perform transcoding faster than the total time of the media file but it's highly dependent on the chipset used. Amlogic chipsets have shown poor performance to transcode HD streams with 50 frames per second and, looking at the obtained results, the system running on these chipsets would not be able to perform live-transcoding of these types of channels' characteristics. Another interesting result was the relation between bitrate and time to transcode, in Snapdragon and RTD 1296 chipsets there seems to be no relation whatsoever but, in the Amlogic chipsets, this correlation is evident and low bitrates impact heavily the system performance.

The obtained results, when the system was runnning on Amlogic chipsets, also shown the worst PSNR values as can be seen in plot 6.2. The PSNR values for 8-bit pixels normally vary between 30 dB and 50 dB, the higher the PSNR value the less noise was introduced in the encoding

Stream	Resolution	Frame Rate	Codec	Bitrate
NHK Worl	1920x1080	25 Hz	H264	10 Mbit/s
3SAT	1280x720	50 Hz	H264	15 Mbit/s
MCS	3840x2160	50 Hz	H265	30 Mbit/s

Table 6.2: Video Streams Characteristics

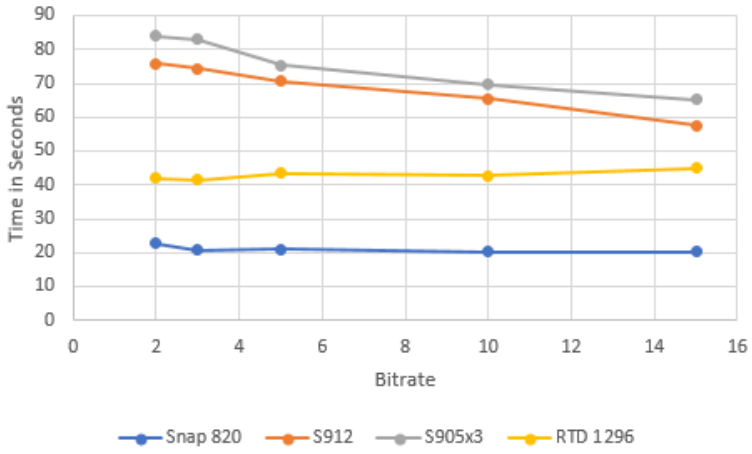


Figure 6.1: Time to transcode an HD stream with 74 seconds and different bitrates.

process. Re-encoding information always adds noise and the smaller the bitrate the higher the noise.

In the obtained results there's a chipset that stands out, Amlogic S912, this chipset cannot handle encoding of higher resolutions than SD and this impacts the system without him knowing something is wrong.

In order to justify the reason for the obtained PSNR results, random frames were extracted from each transcoded media and the same frame is represented in Figure 6.3. The Figure shows the same frame transcoded across different chipsets:

- Sub-figure 6.3a shows the poor quality results obtained with the Amlogic S905X3, in HD resolution. The consumer can notice the macro-blocks in the scene and the channel brand cannot be read which is particularly strange since it is a static region across all frames and therefore, should not be hard to predict.

- The results obtain with Amlogic S912 are shown in sub-figure 6.3b, since the encoder maximum buffer size is not correct, the frame looks replicated along the X and Y-axis.

- In sub-figures 6.3c and 6.3d are shown, respectively, the results obtained with Snapdragon 820 and RTD 1296 chipsets. The quality is low but no artifacts neither macro-blocks appear in the frames, the results obtained are very identical across both chipsets.

In summary, the proposed solution showed different possible scenarios of outcomes when being used in different hardware devices. The performance and output quality of this system will always be dependent on the performance and quality of the hardware vendor. However, one chipset tested, RTD 1296, proved the initial motivation to implement a low cost transcoder is possible.

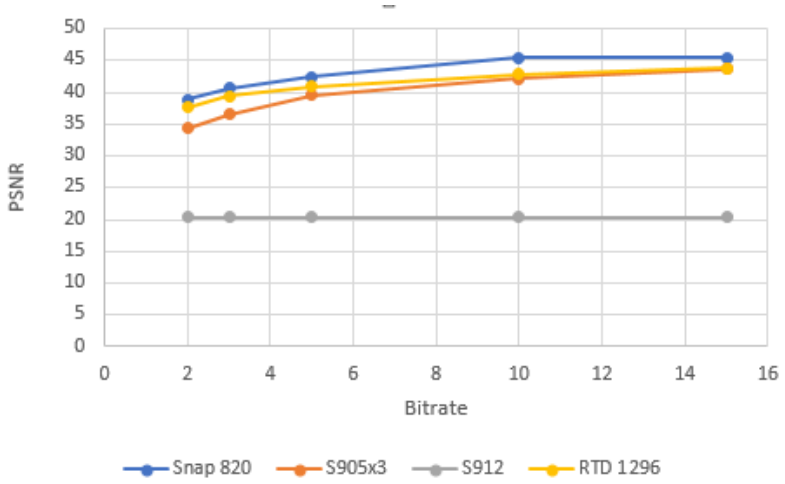


Figure 6.2: PSNR values obtained with different encoding bitrates.

6.2.2 Bitrate Transcoding with FHD Streams

The second test, to evaluate performance and quality during bitrate transcoding, used an Full-HD stream recorded from a Japanese international news channel. This media file has an average bitrate of 10 Mbit/s and a frame rate of 25 frames per second.

In terms of quality, the results were consistent with the previous observed, the system acts very differently according to device it running on. The PSNR results are plotted in Figure 6.5.

Taking again into account the price, performance, and quality, we can conclude the system can perform live transcoding for bitrate reduction and, the best chipset evaluated was the RTD 1296.

6.3 Resolution Transcoding

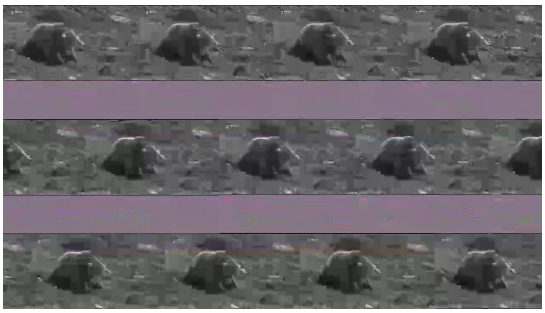
Another use case of the system is to reduce the frame size. Some streams nowadays are encoded in 4K resolutions, this resolution is large and, to minimize its size, another codec is used, HEVC. This codec has better quality, the same stream can be encoded using lower bitrates, the downside, however, is the computation power needed.

AVC codecs do not perform well with 4K resolutions, to change the codec the system also needs to change the resolution. In the next tests, the frame resolution is changed and the output result will be discussed, the last two streams will be taken into account again, 4K resolution transcoding will be illustrated in Codec Transcoding tests since the purpose of the current tests is to evaluate only the change of resolution.

As said during the implementation specification, MediaCodec does not re-sample, in order to perform resolution transcoding, the system must render the frames using OpenGL. Rendering through OpenGL adds a big overhead, in this section the performance overhead and the quality observed from the output stream will be analyzed.

Only two chipsets were used to perform the tests, the results between the snapdragon and the RTD 1296 are very identical so, only the results obtained in the first will be analyzed.

(a) S905D3 Transcoded Frame with bitrate of 2 Mbit/s



(b) S912 Transcoded Frame with bitrate of 2 Mbit/s, HD resolution is not supported

(c) Snapdragon 820 Transcoded Frame with bitrate of 2 Mbit/s

(d) RTD 1296 Transcoded Frame with bitrate of 2 Mbit/s

Figure 6.3: The same frame transcoded across different chipsets

6.3.1 FHD to SD

The first set of tests was performed in the same FHD media file previously used. This time the file has it's only 30 seconds but the characteristics of the stream are the same.

In Figure 6.6 there's a representation of the performance results obtained with the Snapdragon 820. The first test consisted of performing a transcoding operation to register a base time, the stream characteristics were all preserved. With the base time annotated, the same test was performed with OpenGL rendering and, the obtained results, indicate that the overhead added by OpenGL is greater than 50%. The follow-up tests were simulations of real case scenarios, the third test for example, consists of performing resolution transcoding but persisting the other stream characteristics.

In terms of quality, the solution running in the Snapdragon 820 was not the best one, a lot of artifacts were introduced in the stream when the resolution was changed. An example of those artifacts is shown in Figure 6.7. Maybe some alterations in the OpenGL could be performed to minimize the artifacts or some sort of filter could be applicable.

In terms of performance, the the Amlogic S905x3 chipset, can handle encoding of 25 fps in live-transcoding situations and, a particular result observed, was the difference between rendering the frame with and without OpenGL - the results are almost identical which means the bottleneck of the solution in this chipset is not the rendering but the encoding process.

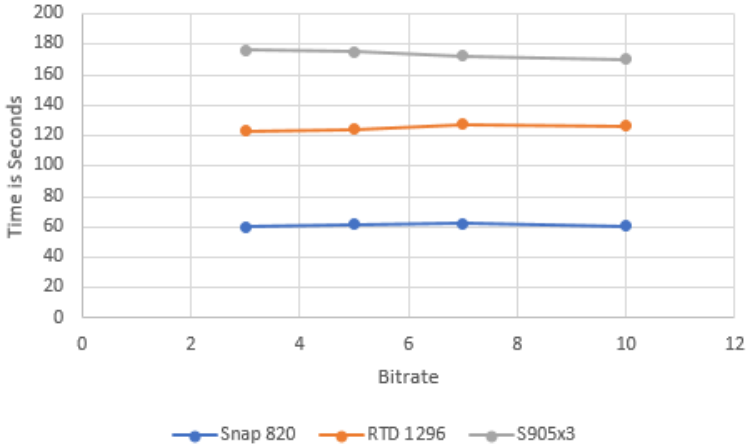


Figure 6.4: Time to transcode an HD stream with 180 seconds and different bitrates.

The transcoding resolution has different outcomes in different chipsets. Some outcomes showed poor results when downsizing the frame resolution, more investigation must be performed to evaluate whether the usage of a different filter can reduce the artifacts in the output stream.

6.3.2 HD to SD

The next tests performed were used to validate the results obtained in the last sub-section. This time, the system will perform resolution transcoding from an HD stream, the same as the one used in bitrate transcoding. To evaluate the performance of the Amlogic chipset in real-world use cases, tests will also be performed with half the frame rate of the original stream.

The tests were performed this time in the RTD 1296 chipset and compared with the obtained results of the Amlogic S905x3 chipset. The system had better performance in the RTD 1296 chipset, however, in terms of quality, better results were observed in the Amlogic chipset and, if the frame rate is dropped in half, the system can theoretically perform live transcoding in this chipset.

In summary, the system can perform resolution transcoding but the current solution is still highly dependent on the performance of the chipset encoder and the quality of resizing the frame with OpenGL. In order to achieve more consistent results, more effort must be applied during pixel interpolation to try to minimize the artifacts shown in the output media file.

6.4 Codec transcoding

HEVC codec has been gaining popularity in recent years, it requires more processing power but, outputs better results with lower bitrates when compared with AVC codec. Being a recent codec means older consumer clients have no support for this codec which means this system can be used to mitigate those problems. In this section, several tests were conducted to evaluate two use

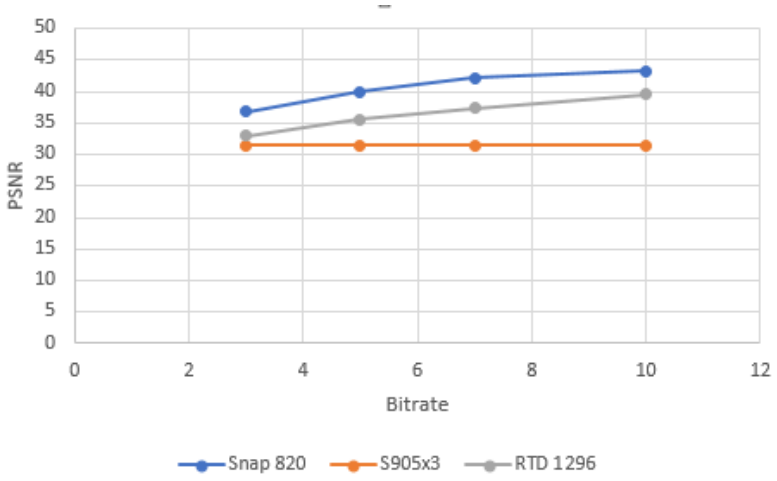


Figure 6.5: PSNR values obtained with different encoding bitrates.

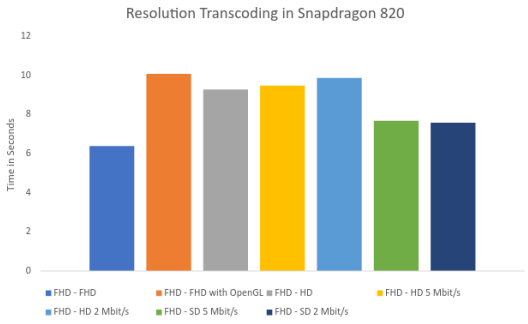


Figure 6.6: Resolution transcoding with snapdragon 820.

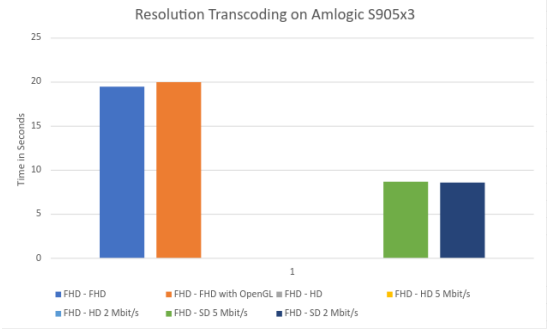


Figure 6.7: Resolution transcoding with amlogic S905x3.

cases: AVC and HEVC quality difference when encoding with low bitrates and transcoding HEVC streams into AVC streams with lower resolution.

Only one chipset was used to evaluate the codec transcoding, the Amlogic S905x3. To test the first use case, several tests were performed to evaluate the performance and quality of the output stream. In terms of performance, it was very similar to the AVC encoder, the system performed well when the frame rate was about 25 frames per second but with higher frame rates, it would not be feasible for live streaming. Quality, in turn, has bigger differences. As can be seen in Figure 6.8, the HEVC codec (6.8b) performed much better than the AVC, no macro-blocks are perceived in the frame.

The next use case tests were performed to evaluate if the system could transcode 4K stream encoded with HEVC codec into a smaller stream encoded with AVC. The tests obtained shown great results, the output stream was smooth without any artifact. The only downside was the quality of the stream due to the new low bitrate but, as proved before, the system outputs poor quality at low bitrates or high frame rates in this chipset. In Figure 6.9 there's an example of an output stream.

These results prove the possibility of the system to support the two illustrated use cases. These

(a) Frame from a video stream encoded with AVC codec.

(b) Frame from a video stream encoded with HEVC codec.

Figure 6.8: Comparison between AVC and HEVC codecs encoding capabilities

use cases are very important and open several practical applications. The first use case, for example, the possibility of feasibly encode with HEVC codec and obtain smaller encoded streams with high quality, could allow the system to further encode IPTV channels to be sent in a CDN networks, where size is a big limitation. The second use case, being able to transcode HEVC encoded stream into an AVC, allows the system to transcode the media stream into a new format that is supported by more end-users that otherwise, could not consume that content.

6.5 Live Transcoding

The next set of tests were created to evaluate how the system handles media content in live streams. The most important aspect of live-transcoding is performance, that's why quality tests are not performed in this section, of course, quality is important for a playback consumer but the same results will be achieved whether the system is performing live transcoding or offline content transcoding. Besides, trying to compare the quality of different chipsets during live transcoding is not accurate, the same stream can have different characteristics along the playback and different results would be registered. That's the main reason behind focusing on offline content to evaluate the quality of each chipset and live content for the overall quality of the system. In offline content we evaluated performance based on the time the system took to transcode a fixed-length media stream, however, in live transcoding this metric does not hold. In live transcoding, the system does not have a fixed length media stream to read from, data is only available to be parsed once new packets are read from the network, which means time to transcode is not relevant since the system cannot outperform the live content in the network.

One heuristic that can be used to evaluate live transcoding is how quickly the system handles new data and if the system can efficiently transcode that data in minimal time. MediaSource layer implements data buffering, once data is read from the source and parsed, it is held in buffers until the decoder requests data to be decoded and, in order to fetch new data, the encoder must encode the previously decoded data. We can use the amount of data held in MediaSource to evaluate whether the system is performing well, in which case, the amount of data buffered is stable. However, if the buffered data keeps increasing, it means the system cannot respond effectively.

Figure 6.9: Frame decoded from a 4K stream and encoded into a SD stream.

To perform the described tests, the amount of data in the buffers will be registered every 5s and, some notes about the status of the live transcoding output from a playback consumer's perspective, are mentioned. The tests will represent real-world desired use cases.

It is worth mentioning that performance tests were already conducted before, this section discusses only the different results obtained transcoding live content.

The test results of live transcoding are illustrated in Figure 6.10. A system color was used to identify the overall quality and performance of real-world use cases, red means the stream is not streamable, yellow means the stream is streamable however there are some quality artifacts and green for success.

The tests conducted in the RTD 1296 chipset consisted in transcoding one audio stream and one video stream, the characteristics of the transcoding process for video streams are presented in the header of the table in Figure 6.10. The results obtained in live transcoding are consistent with the ones previously obtained. The system has a good performance and the only drawbacks are the artifacts introduced while transcoding the frame resolution.

The tests conducted in Amlogic S905x3 were also consistently, the system could not encode with high frame rates in useful time and the quality is not acceptable when using low bitrates. One interesting result was performing frame resolution with low frame rates as output, the chipset performed well and the quality is good as long as the bitrate is high enough.

6.6 Summary

The system is highly dependent on the chipset, this chapter proved there are several use cases the system can handle but not all in the same board. For frame rate resolutions, the system has shown better results in the Amlogic chipset however, in terms of performance and bitrate transcoding the system had better results when using the RTD 1296 chipset.

	- 4K to HD - HEVC to AVC - 20 to 5 Mbit/s - 50 to 25 fps	- 4K to SD - HEVC to AVC - 20 to 3 Mbit/s	- 4K to SD - HEVC to AVC - 20 to 3 Mbit/s - 50 to 25 fps	- FHD to HD - 10 to 5 Mbit/s	- FHD to SD - 10 to 3 Mbit/s	- HD to HD - 15 to 4 Mbit/s	- HD to HD - 15 to 4 Mbit/s - 50 to 25 fps	- HD to SD - 15 to 2 Mbit/s	- HD to SD - 15 to 4 Mbit/s
RTD 1296	Not Supported	Not Supported	Not Supported						
Amlogic S905D3									

Figure 6.10: Tests performed using live content

Based on the obtained results, the system can be used to perform a big set of operations to mitigate real-world problems found in IPTV systems. It's possible to change the codec to support legacy content or, downsize the media content length to cheaply distribute it. The possibilities are endless but, it is up to the user to evaluate which hardware will perform better in his use cases.

Chapter 7

Conclusion and Future Work

To obey the growing demand for multimedia content consumption, several technologies have arisen to mitigate specific problems behind it. One of those technologies is IPTV solutions which allow media content distribution over IP networks. The goal of this thesis is to mitigate some barriers to those technologies. One of those barriers is to support new technologies in legacy hardware another is high consumption of bandwidth.

There are already in the market several alternatives to mitigate the referred limitations one of them is transcoding. By decoding the media content a transcoder can change its characteristics and re-encode it. However, this operation is expensive and there are several challenges such as having to parse the media content from its container, implement the necessary operations to change the data's characteristics prior to being encoded and encode the media content into a container.

The proposed solution is a hardware-assisted transcoder capable of mitigating those barriers. This solution can be implemented in any Android Set-Up box and is heavily optimized for the Android Platform and media operations. To mitigate the problems mentioned, this solution uses the Exoplayer demultiplexer and the FFmpeg multiplexer capabilities. To decode and encode media streams, the system uses Android abstraction to OpenMAX, MediaCodec and the data processing is performed in CPU/GPU:

- Changing frame size is performed in GPU with OpenGL, once the frame is decoded a Texture Render is used to render it into a new size.

- Changing audio sample rate is performed in CPU using linear, cubic, sinc with original coefficients or sinc with revised coefficients resamplers.

- To reduce the number of frames per second, the current solution drops decoded frames prior to being encoded.

- Changing the stream bitrate can be done by configuring the output bitrate of the encoder.

The outcome solution is a transcoder capable of performing must of the operations related to IP Networks. The system can be used to transcode live content or create live content from different sources and protocols. It is possible to stream media content into the IP network from a local file or a downloaded file in the server and, with the support of new content distribution mechanisms, this system can also integrate other services into IPTV.

Performance. In terms of performance, by implementing the transcoder with hardware assistance clearly outperforms any software implementation. The system handled the load perfectly and even the less powerful chipset used, Cortex A53, was at 50% load when transcoding a 4K stream to HD resolution. From the tests performed, the system is heavily dependent on the DSP performance, however, depending on the stream an in the platform, the user can have acceptable performance in digital live transcoding of real-world scenarios. Several optimizations were made to reach the best possible scenario and one of the main concerns was data processing and communication, the system is optimized to avoid at all cost copying data from one memory location to another and, due to the overhead from Java Virtual Machine, several memory-related procedures were ported to Android Native Layer.

Cost. Cost is the biggest motivation behind all work performed previously, Nonius options to transcode were limited to AVC video streams and Full HD and the cost per service was roughly 100$. With the proposed solution, the available range of operations is much bigger and personalized to each use case, the user can choose hardware that best suits its project requirements with costs as low as 20$ per service (considering one Set-Up Box per service).

Limitations. The system has its own limitations, the principal limitation is its dependency on the Android Platform. Although the system architecture is modular, some basic mechanisms, for example, multi-threading, are not. Another limitation is due to the hardware instability, as seen from the conducted tests, many chipsets are not well supported and that is reflected in the absence of features announced by the vendors. Finally, although the available hardware-assisted codecs perfectly fit this transcoder scenario, they are very limited and it is hard to find a set-up box with encoders other than AVC or HEVC.

7.1 Future Work

There's still much work ahead, the main focus in the future will be stability and fault tolerance. The fact that the continuous involve parts of the system, such as demultiplexer and multiplexer, were ported from other projects allows the system to keep up with the market trends. However, there are still some features missing:

- Optimize output load control, to avoid bandwidth and consumer overload. The current solution is not perfect and sometimes the media content playback is affected after transcoding.

- More audio processing operations.

- Apply filters to frames. Since the system already takes advantage of OpenGL to render frames, it could apply some OpenGL shaders to minimize the artifacts of resolution transcoding or to insert new content into the video stream.
- Increase multiplex operations and better support. Currently, the integration with FFmpeg is very limited.
- Increase Performance Continuously.
- Parallel services transcoding. The current solution can only transcode one service at each time, even though a service can be composed by several media streams. In the future, the system could balance the load of transcoding different services.
- Automate the process of evaluating the performance/quality of each chipset and the process of selecting which tracks should be transcoded and what should be the output format.

References

[AS15] M. Arrivukannamma and J.G.R. Sathiaseelan. A study on codec quality metric in video compression techniques. *2015 International Conference on Innovations in Information, Embedded and Communication Systems (ICIIECS). Proceedings*, 2015.

[BBG15] Shraddha Bhatte, Dr. J. W. Bakal, and Madhuri Gedam. Privacy protection for video, image, text transmission. *International Journal of Innovative Research in Computer and Communication Engineering*, 2015.

[dve20] Iptv system for hotel(ip). http://www.hkiptv.com/m/soluView.asp?nid=16, Accessed January 3, 2020.

[ES06] Hazım Ekenel and Rainer Stiefelhagen. Analysis of local appearance-based face recognition: Effects of feature selection and feature normalization. *Computer Vision and Pattern Recognition Workshop, 2006*, pages 34– 34, 07 2006.

[Exo20] ExoPlayer. Exoplayer. http://exoplayer.dev, accessed January 3, 2020.

[GCDCM+13] Rosario Garrido-Cantos, Jan De Cock, José Martínez, Sebastiaan Van Leuven, Pedro Cuenca, and Antonio Garrido. Low complexity transcoding algorithm from h.264/avc-to-svc using data mining. *EURASIP Journal on Advances in Signal Processing*, 2013, 04 2013.

[GH15] Zhu Guqiao and Song Hao. Comparison and analysis of mpeg-dash and hls adaptive streaming delivery technology. *Telecommunications Science*, 31(4):2015096 (5 pp.), 2015.

[ISO19] Information technology — generic coding of moving pictures and associated audio information. Standard, June 2019.

[JOY+13] Minseok Jang, Hyeontak Oh, Jinhong Yang, Jun Kyun Choi, Keuneun Kim, and Ilkwon Cho. Implementation of continuous http live streaming using playback position request mechanism in heterogeneous networks. *2013 15th International Conference on Advanced Communication Technology (ICACT)*, pages 990 – 3, 2013.

[KCLR15] Eunjung Kwon, Shun-Shim Chun, Young-Tae Lee, and Won Ryu. A cloud transcoder using download cache scheme. *2015 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 801 – 3, 2015.

[kn:07] *IPTV: The Ultimate Viewing Experience*. John Wiley Sons, Ltd, 2007.

[kn:19a] Openmax - the standard for media library portability. Available at: https://www.khronos.org/openmax/, Accessed July 3, 2019.

[kn:19b] Jpeg image compression systems. Available at: https://www.ece.ucdavis.edu/cerl/reliablejpeg/compression/, accessed July 5, 2019.

[LBC06] D. Lefol, D. Bull, and N. Canagarajah. Performance evaluation of transcoding algorithms for h.264. *IEEE Transactions on Consumer Electronics*, 52(1):215–222, Feb 2006.

[Lim13] Y. Lim. Mmt, new alternative to mpeg-2 ts and rtp. In *2013 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*, pages 1–5, June 2013.

[LLRC09] G. M. Lee, C. S. Lee, W. S. Rhee, and J. K. Choi. Functional architecture for ngn-based personalized iptv services. *IEEE Transactions on Broadcasting*, 55(2):329–342, June 2009.

[LM08] Francesca Lonetti and Francesca Martelli. *Temporal Video Transcoding for Multimedia Services*, pages 149–179. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[LW09] Min Li and Bo Wang. Hybrid video transcoder for bitrate reduction of h.264 bit streams. *2009 Fifth International Conference on Information Assurance and Security (IAS)*, vol.1:107 – 10, 2009.

[Men19] Ana Carolina Menezes. *Mpeg-DASH*. Available at: https://www.gta.ufrj.br/ensino/eel879/vf/mpeg-dash, Accessed July 3, 2019.

[OM06] Jose Oliver and Manuel P. Malumbres. From lossy to lossless wavelet image coding in a tree-based encoder with resolution scalability. In José Francisco Martínez-Trinidad, Jesús Ariel Carrasco Ochoa, and Josef Kittler, editors, *Progress in Pattern Recognition, Image Analysis and Applications*, pages 198–207, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[PKK+09] A. Prasad, K. Krishnan, Karthika, Parvathy, and R. K. Megalingam. Low power losssless compression of real time mpeg4 video encoding and decoding using vhdl and matlab. In *2009 2nd IEEE International Conference on Computer Science and Information Technology*, pages 408–412, Aug 2009.

[Ric04a] Iain Richardson. H.264 and mpeg-4 video compression : video coding for next-generation multimedia. *SERBIULA (sistema Librum 2.0)*, 01 2004.

[Ric04b] Iain E. G. Richardson. *Video Coding Concepts*. John Wiley Sons, Ltd, 2004.

[sKLK03] Chang sung Kim, Qing Li, and C.-C.J. Kuo. Fast intra-prediction model selection for h.264 codec. *Proceedings of the SPIE - The International Society for Optical Engineering*, 5241(1):99 – 110, 2003.

[SP11] I. Sodagar and H. Pyle. Reinventing multimedia delivery with mpeg-dash. volume 8135, pages 81350R (7 pp.) –, USA, 2011.

REFERENCES

[SVT98] G. Sharma, M. J. Vrhel, and H. J. Trussell. Color imaging for multimedia. *Proceedings of the IEEE*, 86(6):1088–1108, June 1998.

[Tra19] Armin Trattnig. Fun with container formats. Available at: https://bitmovin.com/fun-with-container-formats-3, Accessed July 3, 2019.

[VCH03] A. Vetro, C. Christopoulos, and Huifang Sun. Video transcoding architectures and techniques: an overview. *IEEE Signal Processing Magazine*, 20(2):18–29, March 2003.

[VGJ14] B. Veselinovska, M. Gusev, and T. Janevski. State of the art in iptv. In *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 479–484, May 2014.

[VLDCVW+11] S. Van Leuven, J. De Cock, G. Van Wallendael, R. Van de Walle, R. Garrido-Cantos, J.L. Martinez, and P. Cuenca. A low-complexity closed-loop h.264/avc to quality-scalable svc transcoder. pages 6 pp. –, Piscataway, NJ, USA, 2011.

[Wan10] L. Wang. Bit rate control for hybrid dpcm/dct video codec. *IEEE Transactions on Circuits and Systems for Video Technology*, 4(5):509 – 20, 1994/10/.

[WZT07] GuoZhong Wang, HaiWu Zhao, and Guowei Teng. Usage of mpeg-2 to avs transcoder in iptv system. In Horace H.-S. Ip, Oscar C. Au, Howard Leung, Ming-Ting Sun, Wei-Ying Ma, and Shi-Min Hu, editors, *Advances in Multimedia Information Processing – PCM 2007*, pages 65–70, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[YLB16] J. Yoon, P. Liu, and S. Banerjee. Low-cost video transcoding at the wireless edge. In *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 129–141, Oct 2016.

[ZSAY00] Zhaohui Cai, K. R. Subramanian, Aidong Men, and Yong Ji. A risc implementation of mpeg-2 ts packetization. In *Proceedings Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region*, volume 2, pages 688–691 vol.2, May 2000.

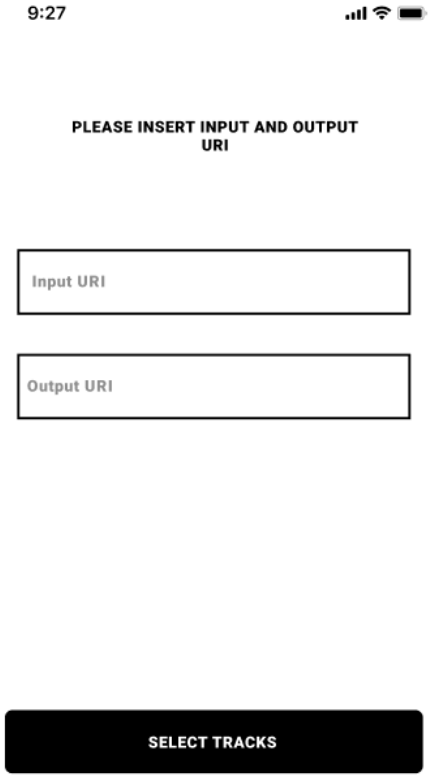
[ZXW18] Chi Zhang, Bo Xiao, and Hanli Wang. Highly parallel acceleration of hevc encoding on arm platform. *2018 IEEE Fourth International Conference on Multimedia Big Data (BigMM)*, pages 1–6, 09 2018.

REFERENCES

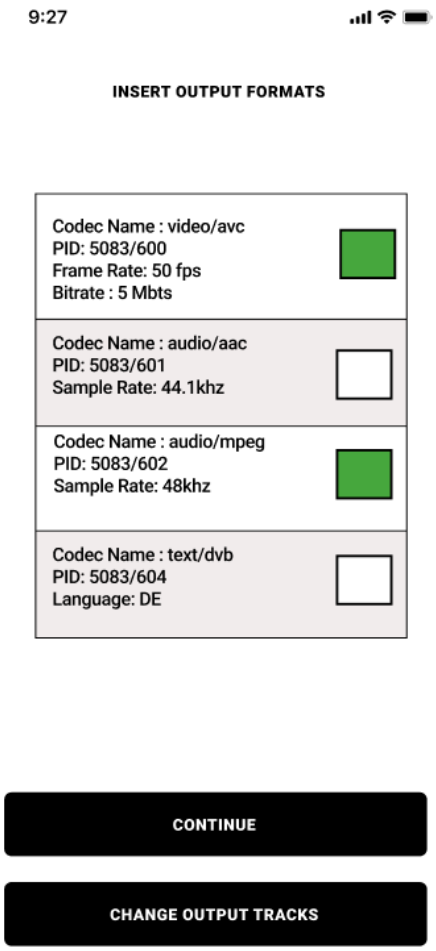
Appendix A

Prototype

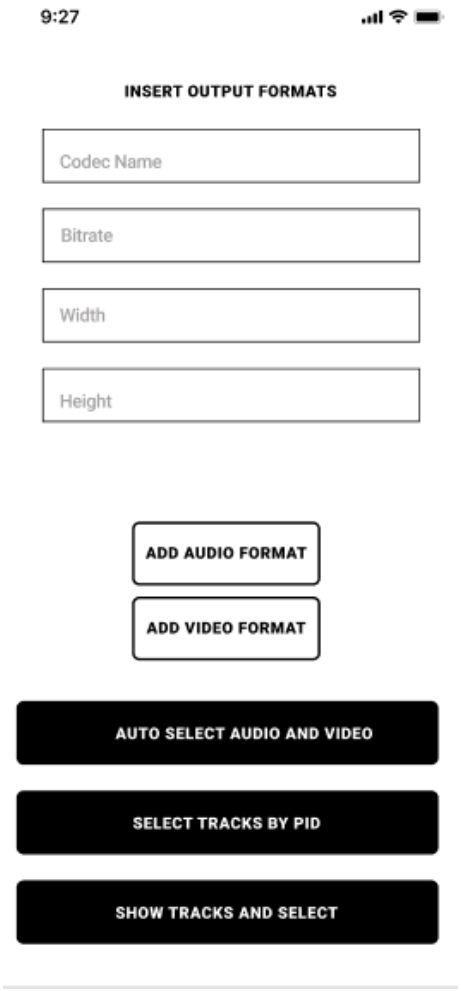
A.1 Solution Client GUI



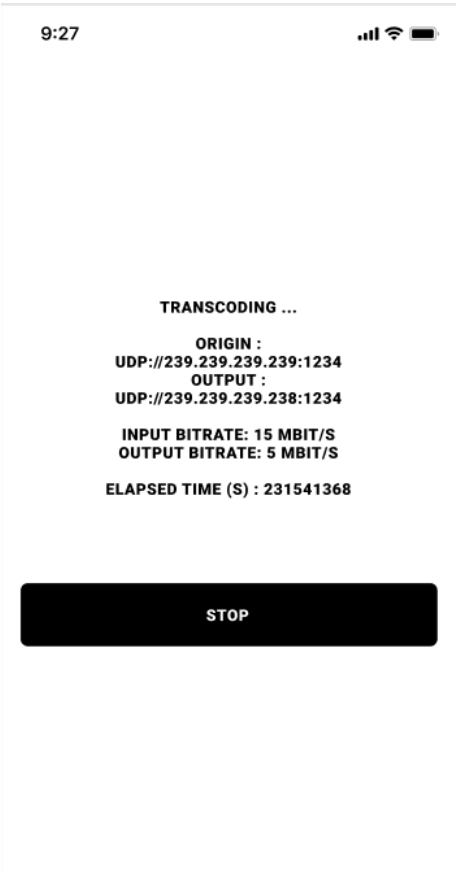
(a) Selecting input and output Uri.



(b) Select Tracks



(a) Selecting Output Formats.



(b) System Running Information