Faculdade de Engenharia da Universidade do Porto

Mobile Context-Aware Multimedia Application

Rui Pedro Ferreira Pinto

Thesis dissertation carried out under the Master in Electrical and Computers Engineering, Major in Telecommunications, Electronics and Computers

Supervisor: Maria Teresa Andrade (PhD)

June 2013

Resumo

O contexto das comunicações multimédia tem evoluido significativamente nas últimas décadas, originando uma enorme difusão de vários formatos de conteúdos e novas tecnologias nos dispositivos usados pelos clientes. O uso de dispositivos móveis, como por exemplo, o smartphone, associado ao acesso á Internet tornou-se ubíquo, possibilitando o consumo de conteúdos multimédia associado ás nossas rotinas diárias. Estes dispositivos podem ser bastante diferentes entre si, em termos de hardware, poder computacional, sistema operativo e software instalado. Existe igualmente bastante diversidade em relação ao conteúdo multimédia, pois este pode ser composto por diferentes combinações (imagem, áudio, video, texto, gráficos, animações, etc), com uma grande variedade de formatos de codificação á disposição e um largo intervalo de possíveis valores para bit rate, resoluções espaciais e temporais, qualidade e até dimensões adicionais (*2D* versus *3D*).

Neste cenário de heterogenidade, é difícil encontrar um dispositivo móvel capaz de descodificar e reproduzir todas as possíveis variações do conteúdo com igual nível de qualidade. Para além disso, as expectativas dos consumidores tem vindo a aumentar, exigindo serviços personalizados ás suas necessidades e preferências, sem ter em consideração o dispositivo que está a usar, as condições da rede a que está sujeito ou as características do ambiente fisico em que se encontra.

O principal objectivo desta dissertação é estudar e desenvolver soluções para esse desafio eficientemente, adaptando os conteúdos multimédia consoante a informação de contexto, respondendo dinamicamente a diferentes soluções, consições e restrições.

Com base no estudo realizado anteriormente no âmbito do estado da arte sobre technologias e protocolos existentes, um sistema cliente-servidor foi proposto para desenvolvimento, oferecendo a clientes *Android* um serviço de streaming adaptável de conteúdos multimédia. A solução adotada usa a recente norma *MPEG-DASH* para adaptar dinamicamente o bit rate dos conteúdos multimédia enviados pelo servidor para o cliente, de acordo com as condições de rede, nomeadamente a largura de banda disponível. Para além disso, um agente de contexto a correr no cliente recolhe informações de vários sensores que integram o dispositivo *Android*

usado. Essas informações serão usadas para caracterizar o meio ambiente e decidir se uma adaptação de conteúdos é desejável para maximizar a qualidade de experiência do cliente.

Abstract

The field of multimedia communications has evolved tremendously in the last decade, leading to a widespread of multiple content formats and client devices' technology. The use of mobile terminals, such as smartphones, together with Internet access have become ubiquitous, enabling the consumption of multimedia content associated to most of our daily routines.

These multimedia-enabled mobile devices can be very different from each other in terms of hardware, computational power, operating system and software installed. There is also a great heterogeneity in relation to the multimedia content itself, which can be composed of different combinations of diverse media (image, audio, video, graphics, animation, text, etc.), with a great variety of encoding formats available to choose from and large range of possible values for bit rate, spatial and temporal resolutions, quality or even additional dimensions (*2D* versus *3D*).

Within this heterogeneous scenario, it is hard to find a mobile phone capable of decoding and presenting all possible content variations with the same quality level. Additionally, the expectations of consumers have risen, now demanding personalized services matching their needs and preferences, regardless the type of terminal being used, the conditions of the available network connection or the characteristics of the surrounding natural environment.

The goal of this dissertation was to study and develop solutions to that challenge in an efficient way, adapting the multimedia content considering the usage context, responding dynamically to different situations, conditions and restrictions.

Based on the study conducted on relevant existing technologies and protocols at the forefront of the state-of-the-art, a client-server system was proposed and developed, offering to *Android* clients a transparent and adaptable multimedia streaming service. The adopted solution makes use of the emergent standard *MPEG-DASH* to dynamically adapt the bit rate of the streamed media according to the network availability. Additionally, a context agent running on the client, collects information from varied built in sensors of the *Android*

smartphone to characterize the surrounding environment and decide if further adaptation is desirable to maximize the quality of experience of the user.

Acknowledgments

I would like to express my gratitude to my supervisor Maria Teresa Andrade for her guidance along the development of this project.

Thanks to all my good friends and family for the support and motivation in these past years of my academic life.

Contents

List of Figures

List of Tables

Abbreviations and Symbols

Abbreviations (alphabetical order)

2D	Two-dimensional space
3D	Three-dimensional space
3GP	Multimedia container format defined by the Third Generation Partnership Project
A/V	Audiovisual
ADT	Android Development Tools
AVI	Audio Video Interleaved
API	Application Programming Interface
ARM	Industry’s leading supplier of microprocessor technologies
CC/PP	Composite Capability/Preference Profiles
CDN	Content Delivery Network
CPU	Central Processing Unit
DASH	Dynamic Adaptive Streaming over HTTP
DIA	Digital Item Adaptation
DCO	Delivery Context Ontology
DRM	Digital Rights Management
FLV	Flash Video
GPAC	Open Source Multimedia Framework
GPS	Global Positioning System
H.264	Standard for video and audio compression
HD	High-Definition
HDS	HTTP Dynamic Streaming
HLS	HTTP Live Streaming
HSDPA	High-Speed Downlink Packet Access
HTML	HyperText Markup Language
HTML5	Fifth revision of HTML standard
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure

Hz	SI unit for frequency
IDE	Integrated Development Environment
IIS	Internet Information Services
ISO	International Organization for Standardization
ISO BMFF	ISO Base Media File Format
JDK	Java Development Kit
JPEG	Joint Photographic Experts Group
JPEG 2000	Standard for image compression
JRE	Java Runtime Environment
JVM	Java Virtual Machine
Lux	SI unit of luminance
M4V	Video file format developed by Apple and is very close to the MP4 format
Mbps	Mega bit per second
MIME	Multipurpose Internet Mail Extensions
MKV	Matroska Multimedia Container
MP	Megapixel, term used to express the number of image sensor elements of digital cameras
MP4	MPEG-4 Part 14
MPD	Media Presentation Description
MPEG	Moving Pictures Experts Group
MPEG-1	Standard for video and audio compression
MPEG-2	Standard for video and audio compression
MPEG-4	Standard for video and audio compression
MPEG-2 TS	MPEG Transport Stream
MOV	File format used natively by the QuickTime framework
MOS	Mean Opinion Score
NAT	Network Address Translation
NDK	Native Development Kit
OMA	Open Mobile Alliance
OS	Operating System
OWL	Web Ontology Language
Pa	Pascal, derived unit of pressure
PDA	Personal Digital Assistant
PES	Packetized Elementary Stream
RDF	Resource Description Framework
RGB	Color space based on the RGB color model
RMVB	RealMedia Variable Bitrate
RTMP	Real Time Messaging Protocol

RTSP	Real Time Streaming Protocol
SDK	Software Development Kit
TCP	Transmission Control Protocol
TS	MPEG Transport Stream
UAProf	User Agent Profile
UDP	User Datagram Protocol
UED	Usage Environment Description
URL	Uniform Resource Locator
VoD	Video on Demand
WAP	Wireless Application Protocol
WML	Wireless Markup Language
WMV	Windows Media Video
WTA	Wireless Telephony Application
XHTML	Extensible HyperText Markup Language
XML	Extensible Markup Language
ZIP	Popular archive file format that supports lossless data compression

Symbols

x	Ambient light level
R	Red component of the RGB color model
G	Green component of the RGB color model
B	Blue component of the RGB color model
P	Sound pressure
$P0$	Reference sound pressure

Chapter 1

Introduction

The project developed in this thesis is intended to study and develop solutions which allow the delivery of adapted content based on different situations, conditions and restrictions. As the title of this thesis suggests, it was developed a video player mobile application. This player is integrated on a bigger system, together with an *HTTP* server. The application was developed on *Android* platform.

Android is the operating system that is growing faster in the mobile *OS* business, millions of new devices are being activated a month. *Android* phones are very popular because of the choices it gives to the client. Imagine powerful phones, made by different manufactures, which can run tens of thousands of applications, but fully customizable. Several manufactures means that monopoly is not of just one company. This can be a good thing, because of the rapid progress made, caused by their competition with each other. Contributions of the open-source *Linux* community and more than 300 hardware, software and carrier partners, makes *Android* phones more powerful and innovated every day. Besides that, his powerful development framework gives developer everything he need to build best-in-class experiences, letting him deploy his apps broadly to hundreds of millions of users across a wide range of devices – from phones, to tablets and beyond.

Due to the huge terminal's heterogeneity on software and hardware, as well as the variety of network and environmental conditions, emerged the need to personalize the service to each client. The special functionality of the video player developed is that it is prepared to play content via adaptive streaming. The player collects usage context information and uses it to adapt the video quality, for a better user experience. The video adaptation consists on requesting different video qualities from the server, using *HTTP* protocol for the requests.

Quality of experience is a subjective measure of the user experience with a service, in this case, a video streaming playback. This end-to-end performance measure is very important for video services, because many factors may highly affect the user experience, such as network, *CPU*, environmental noise and brightness conditions. Usage context information is the collection of measures made to evaluate these conditions. This information is used to adapt the video, resulting in very little rebuffering, fast start time and good experience for both high-end and low-end connections.

1.1. Motivation

Mobile technologies have revolutionized the way people make decisions and live their daily routine. The future of computer technology rests in mobile computing with wireless networking. Everybody will be connected to everybody, constantly exchanging data.

The next generation of smartphones is going to be context-aware. The phone will take advantage of the growing availability of embedded sensors and data exchange abilities. Biosensors will be used to send alerts of emergencies, is there is something wrong with our body functionality, due to an accident. Physical sensors are now being used to locate the user and measure the environmental conditions.

Nowadays, with this incredible flow of online multimedia content, the first natural step is to take advantage of context-aware information to deliver the context the best way possible. The work developed in this dissertation help understand a little better about the power of using the device's resources on a web service, to provide the best quality of experience to everyone.

1.2. Main Goals

- Identification of the *Android* support sensors and other resources, which will be used to measure the smartphone properties, network and environment conditions.
- Acquire and interpret usage context information.
- Develop a smartphone application to present video files based on an adaptive web streaming.
- Develop a web server to store the files to be delivered to the client.
- Use different strategies to decide the best content form to deliver, based on the context information collected.
- Be able to display the video using progressive download.

1.3. Document Structure

The present document is structured as follows:

- Chapter 1: Introduction: brief explanation about the following document and the research, analysis and conclusion of the project developed.
- Chapter 2: Technologies for Multimedia Adaptable Application: description of the research made about *A/V* content adaptation on a streaming session;
- Chapter 3: Android API: description of the *Android API*, the operating system that supports the application developed;
- Chapter 4: MPEG-DASH: description and analysis of the functionalities offered by *MPEG-DASH*, the protocol used to adapt the *A/V* content.
- Chapter 5: System Architecture and Implementation: description and analysis of the project developed.

Chapter 2

Technologies for Multimedia Adaptable Application

2.1. Usage Context Information

The multimedia content adaptation can be done using the context information of each user. The goal is to adapt the content in a smart way, keeping track of the consumer environment, including usage preferences, and automatically modify the content characteristics to satisfy context variations.

2.1.1. Types of Contextual Information

There are several context characteristics to be aware of:

- Terminal capacities – types of decoders, processing, memory and storage capacity, screen dimensions, etc.
- Environment characteristics – lighting conditions, noise level, etc.
- User preferences – favorite media type, content type, visual and auditory impairment, languages/subtitles, etc.
- Network characteristics – bandwidth capacity, delays, errors, etc.

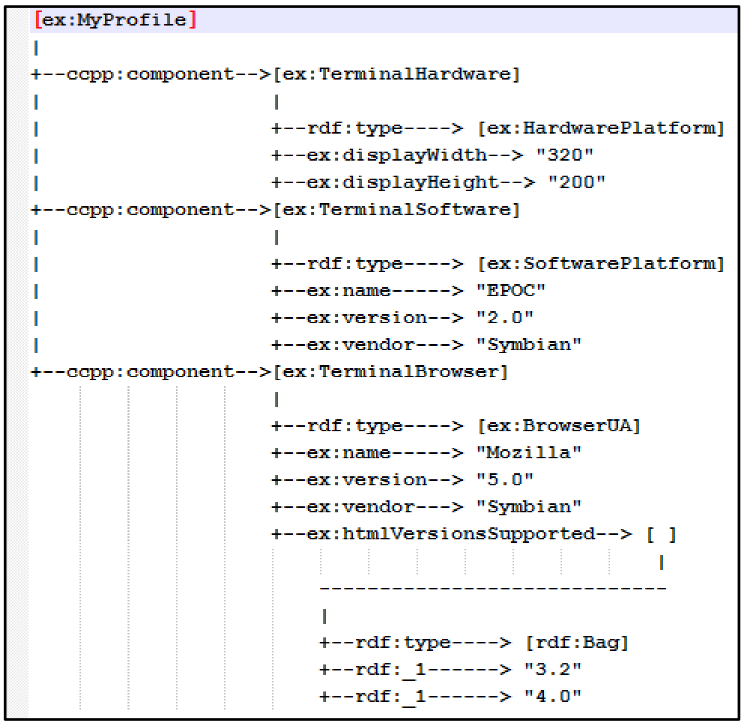
2.1.2. Representation of Contextual Information

2.1.2.1. CC/PP

A *CC/PP* profile [1-5] is a description of the terminal's capacities and preferences of his user. It is used to adapt multimedia content consumed in that terminal.

CC/PP is based in *RDF*, which is a model of metadata description used in web resources.

A *CC/PP* profile is structured on a two level hierarchy: components and attributes. Components are groups of related attributes, like the terminal hardware and software properties. The attributes have values associated to them, used by the server to know the best way to deliver media to the client.



```
[ex:MyProfile]
|
+--ccpp:component-->[ex:TerminalHardware]
|                   |
|                   +--rdf:type----> [ex:HardwarePlatform]
|                   +--ex:displayWidth--> "320"
|                   +--ex:displayHeight--> "200"
+--ccpp:component-->[ex:TerminalSoftware]
|                   |
|                   +--rdf:type----> [ex:SoftwarePlatform]
|                   +--ex:name-----> "EPOC"
|                   +--ex:version--> "2.0"
|                   +--ex:vendor---> "Symbian"
+--ccpp:component-->[ex:TerminalBrowser]
                    |
                    +--rdf:type----> [ex:BrowserUA]
                    +--ex:name-----> "Mozilla"
                    +--ex:version--> "5.0"
                    +--ex:vendor---> "Symbian"
                    +--ex:htmlVersionsSupported--> [ ]
                                                     |
                        -----------------------------
                        |
                        +--rdf:type----> [rdf:Bag]
                        +--rdf:_1------> "3.2"
                        +--rdf:_1------> "4.0"
```

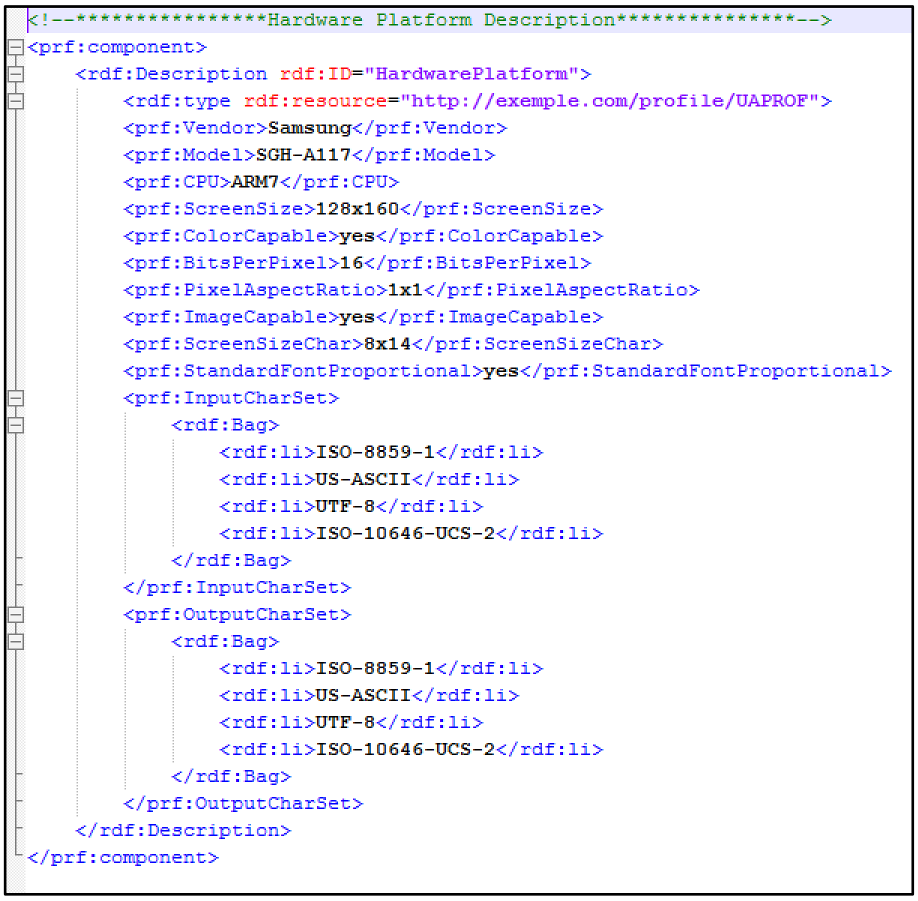
Figure 1 - CC/PP Profile example.

2.1.2.2. UAProf

The *UAProf* [1-5] was defined by *OMA*. It's based on *CC/PP* and is used by multimedia servers to choose the best way to deliver the content to wireless terminals.

In opposition to *CC/PP*, *UAProf* defines a vocabulary to describe characteristics and capacities of the wireless terminal. There are six components:

- *HardwarePlatform* – defines nineteen attributes related to the terminal's model, included the *CPU* and the manufacture. It's also detailed information about the screen and the user interaction, made by attributes related to the keyboard and audio output.
- *SoftwarePlatform* – defines twenty three attributes related to the information about supported file types, preferred languages, operating system and installed software, audio and video codecs and support for downloadable software.
- *BrowserUA* – describes information about the browser, *HTML*, *XHTML* and *JavaScript* characteristics. This is important information when trying to access a web page.
- *NetworkCharacteristics* – defines four attributes related with network information, like security and delay. It also has information about Bluetooth support.
- *WapCharacteristics* – describes information related to *WAP*, *WML* and *WTA*.
- *PushCharacteristics* – describes the behavior of pushed information to the terminal without request. Defines information about maximum size character codification in that information.



```
<!--****************Hardware Platform Description***************-->
<prf:component>
    <rdf:Description rdf:ID="HardwarePlatform">
        <rdf:type rdf:resource="http://exemple.com/profile/UAPROF">
        <prf:Vendor>Samsung</prf:Vendor>
        <prf:Model>SGH-A117</prf:Model>
        <prf:CPU>ARM7</prf:CPU>
        <prf:ScreenSize>128x160</prf:ScreenSize>
        <prf:ColorCapable>yes</prf:ColorCapable>
        <prf:BitsPerPixel>16</prf:BitsPerPixel>
        <prf:PixelAspectRatio>1x1</prf:PixelAspectRatio>
        <prf:ImageCapable>yes</prf:ImageCapable>
        <prf:ScreenSizeChar>8x14</prf:ScreenSizeChar>
        <prf:StandardFontProportional>yes</prf:StandardFontProportional>
        <prf:InputCharSet>
            <rdf:Bag>
                <rdf:li>ISO-8859-1</rdf:li>
                <rdf:li>US-ASCII</rdf:li>
                <rdf:li>UTF-8</rdf:li>
                <rdf:li>ISO-10646-UCS-2</rdf:li>
            </rdf:Bag>
        </prf:InputCharSet>
        <prf:OutputCharSet>
            <rdf:Bag>
                <rdf:li>ISO-8859-1</rdf:li>
                <rdf:li>US-ASCII</rdf:li>
                <rdf:li>UTF-8</rdf:li>
                <rdf:li>ISO-10646-UCS-2</rdf:li>
            </rdf:Bag>
        </prf:OutputCharSet>
    </rdf:Description>
</prf:component>
```

Figure 2 - UAProf example.

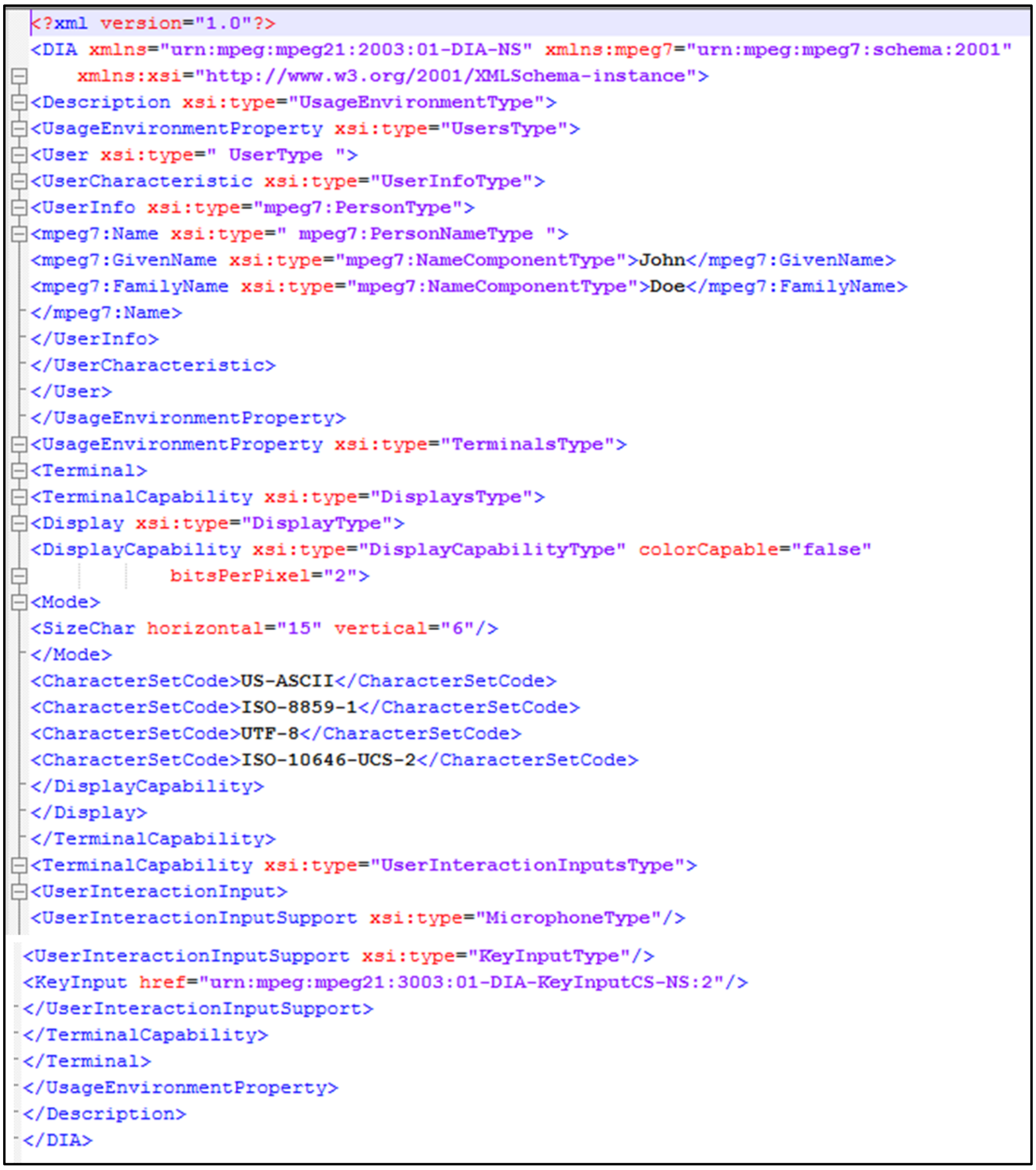
2.1.2.3. UED

UED [1-5] is defined on *DIA*, part 7 of the *MPEG-21*, standard of *MPEG*. *DIA* provide several tools to allow the adaptation of *Digital Items*, defined on part 1 of the *MPEG-21*.

UED is based on *XML* and has a vocabulary organized in four categories:

- User Characteristics – Provide user information, usage preferences and history, presentation preferences, accessibility characteristics like visual and auditory impairments and location information, including the user's movement.
- Terminal Capabilities – Provide information about the terminal such as codec capabilities, content display and audio output capabilities, power/storage characteristics, data input/output facilities, *CPU* capabilities and user interaction possibilities.
- Network Characteristics – Provide the network static and dynamic properties. Static properties can be the maximum capacity, the minimum guaranteed bandwidth the network can provide, information about packets delivery and how errors are handle. Dynamic properties describe currently bandwidth, error and delay.

- Natural Environment Characteristics – Describe physical environmental conditions around the user such as time and location, noise and lighting conditions.



```
<?xml version="1.0"?>
<DIA xmlns="urn:mpeg:mpeg21:2003:01-DIA-NS" xmlns:mpeg7="urn:mpeg:mpeg7:schema:2001"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<Description xsi:type="UsageEnvironmentType">
<UsageEnvironmentProperty xsi:type="UsersType">
<User xsi:type=" UserType ">
<UserCharacteristic xsi:type="UserInfoType">
<UserInfo xsi:type="mpeg7:PersonType">
<mpeg7:Name xsi:type=" mpeg7:PersonNameType ">
<mpeg7:GivenName xsi:type="mpeg7:NameComponentType">John</mpeg7:GivenName>
<mpeg7:FamilyName xsi:type="mpeg7:NameComponentType">Doe</mpeg7:FamilyName>
</mpeg7:Name>
</UserInfo>
</UserCharacteristic>
</User>
</UsageEnvironmentProperty>
<UsageEnvironmentProperty xsi:type="TerminalsType">
<Terminal>
<TerminalCapability xsi:type="DisplaysType">
<Display xsi:type="DisplayType">
<DisplayCapability xsi:type="DisplayCapabilityType" colorCapable="false"
            bitsPerPixel="2">
<Mode>
<SizeChar horizontal="15" vertical="6"/>
</Mode>
<CharacterSetCode>US-ASCII</CharacterSetCode>
<CharacterSetCode>ISO-8859-1</CharacterSetCode>
<CharacterSetCode>UTF-8</CharacterSetCode>
<CharacterSetCode>ISO-10646-UCS-2</CharacterSetCode>
</DisplayCapability>
</Display>
</TerminalCapability>
<TerminalCapability xsi:type="UserInteractionInputsType">
<UserInteractionInput>
<UserInteractionInputSupport xsi:type="MicrophoneType"/>
<UserInteractionInputSupport xsi:type="KeyInputType"/>
<KeyInput href="urn:mpeg:mpeg21:3003:01-DIA-KeyInputCS-NS:2"/>
</UserInteractionInputSupport>
</TerminalCapability>
</Terminal>
</UsageEnvironmentProperty>
</Description>
</DIA>
```

Figure 3 - UED example.

2.1.2.4. DCO

DCO [1-5] is based on *OWL*, providing a set of characteristics that describe the context in which media resources are consumed:

- Supported audio, video and image formats. Colloquial names, *MIME* types, name and version can be given.

- Usable display pixels, supported markups and location provider support.
- Information about the environment (network and location) and user.
- Information about the device's hardware:
  - Battery capacity, current level and whether it's being charged;
  - Bluetooth support;
  - Built-in and extension memory;
  - Cameras;
  - Input/output character sets;
  - Input devices;
  - Network support;
  - Number of soft keys which are programmable;
  - *CPU*;
  - Audio output support;
  - Voice recognition support;
  - Text input type.
- Information about the device's software:
  - Information about the user agent;
  - Whether the delivery context provides support for *Java*;
  - Operating system;
  - Capabilities of the browser.

2.2. Technologies and Protocols for Streaming

2.2.1. Types of Streaming

Streaming allows multimedia content, such as data, audio and video to be delivered to the user by a provider (server). A client media player decodes and presents the media while it is being received. The reproduction starts before the transmission of the entire file is complete.

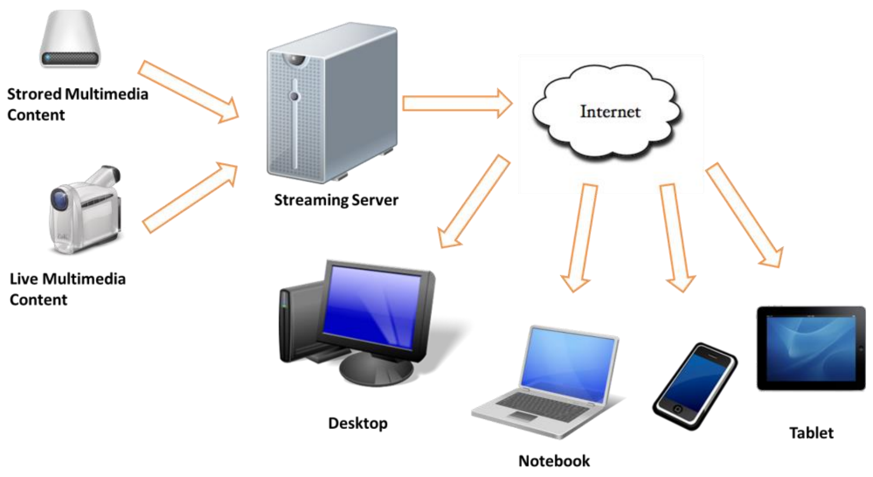


Figure 4 - Generic configuration of a streaming system.

The streaming begins with the codification and the segmentation of the media or signal to be transmitted on Internet. The media can be stored in hard drives and that presented or it can be live, as the media is being produced and presented at the same time.

The client receives the media and reorganizes the segments and packages before introduced them on a buffer. The segments will then be presented, giving the illusion of continuous reproduction. It has to be a compromise between the length of the buffer and the transmission rate. More speed means that buffer length could be shorter. If the transmission is to slow, the number of segments being stored in the file decreases more that the number of segments being presented, to the point where is no more segments stored to be presented, leading to re-buffering, stopping the media presentation while waiting for new segments.

Streaming can be made in 3 different ways, Traditional Streaming, Progressive Download and HTTP Adaptive Stream.

2.2.1.1. Traditional Streaming

Traditional Streaming use sessions to maintain a unique continuous data transmission, different from the other approaches where there are several files to be transmitted, the media data chunked into segments.

The media is coded in small packages and operates over *UDP* or *TCP*. It uses protocols like *RTSP*, *MMS* and *RTMP*.

Unlike *TCP*, *UDP* don't resend requests for missing packages but instead continue with the rest of the file. It's better to have a glitch on the media than the playback to stop while waiting for missing data. The major disadvantage of *UDP* is various problems with firewalls and proxies, blocking some packages and compromising the quality of the reproduction.

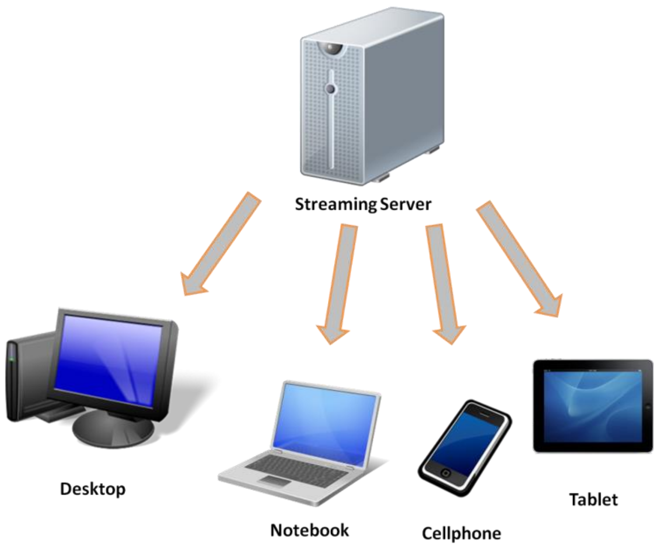


Figure 5 - Server-Client system, using a traditional streaming protocol.

It can begin the playback at any point of the video or skip through it, it makes a lot more efficient use of bandwidth, receiving only the part of the video that will actually be watched and the video file is not stored on a viewer's computer, discarding the data after being played.

2.2.1.2. Progressive Download

Progressive Download deliveries the file over *HTTP*. It is a very simple bulk download of data to the end user's computer. A temporary copy of the file is then stored locally so that the user can access the data every time without downloading it again.

It's supported for most platforms and applications for multimedia content reproduction, allowing playback and download to occur at the same time, before the file is completed.

The major disadvantage of *HTTP* delivery is the inefficiency use of bandwidth as the whole file is delivered despite being or not reproduced till the end.

Clients who support *HTTP 1.1* can specify which part they want of the file and download it, using Byte Range Requests.

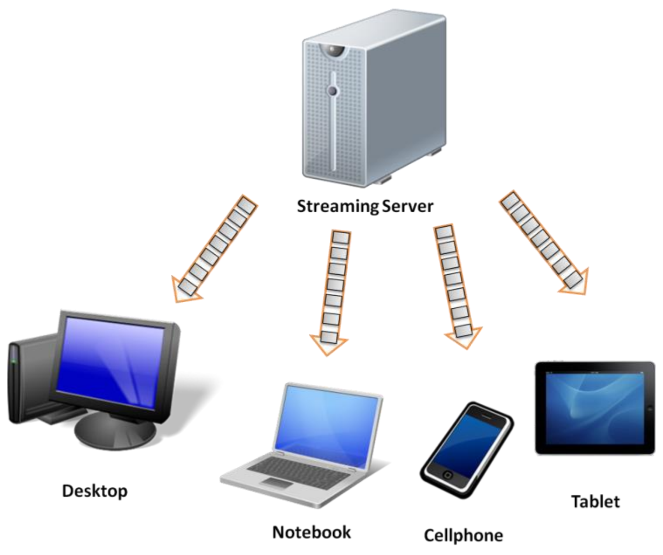


Figure 6 - Server-Client system using a progressive download protocol.

2.2.1.3. HTTP Adaptive Stream

Adaptive Stream is based on Progressive Download, delivering over *HTTP* and storing locally a copy of the file. The difference is that the media stored is segmented in various qualities and the client will request the best for his bandwidth and *CPU* capacity. The content is then organized in a linear sequence for playback, adapting the media as long as it is being played.

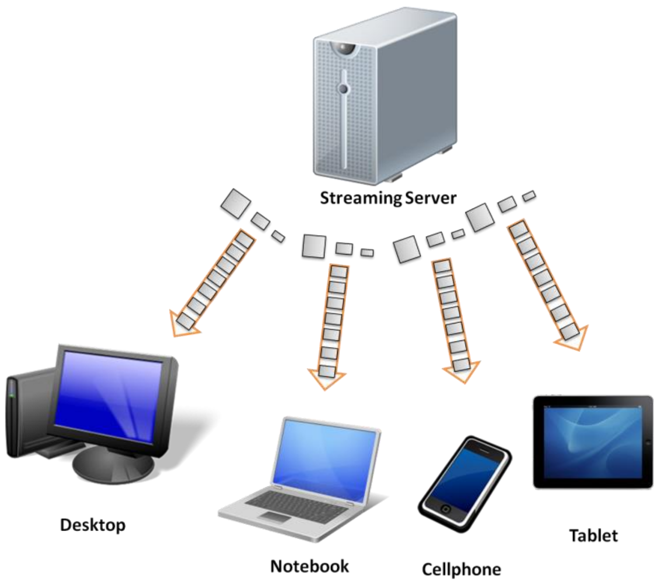


Figure 7 - Server-Client system, using an adaptive streaming protocol.

It has some advantages over Traditional Steaming:

- It doesn't need a special streaming server, it uses *HTTP* caches/proxies, so the server implementation is much cheaper. It takes only a simple web server for this.
- It can dynamically adapt to bad network conditions.
- It allows every user receive media with adequate quality for his network and *CPU* capacities.

2.2.2. HTTP Adaptive Stream Protocols

2.2.2.1. Apple – HLS

HTTP Live Streaming [6] is a media streaming communications protocol based on *HTTP,* created by *Apple Inc*. It supports live or pre-recorded audio and video.

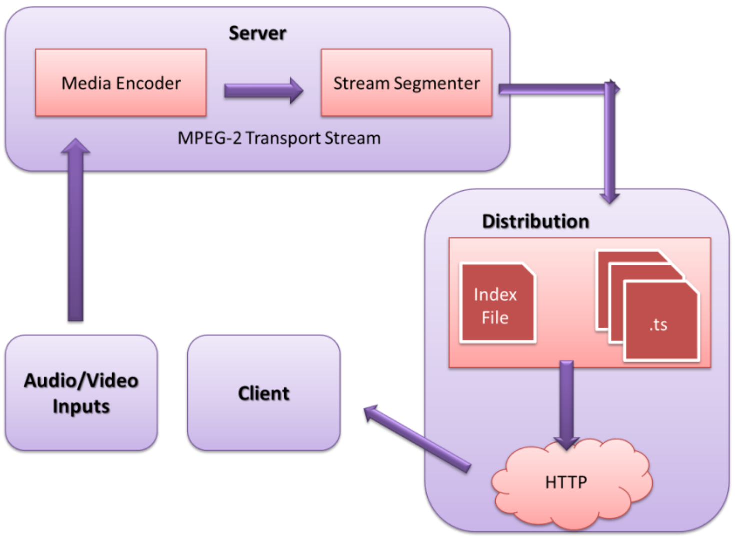


Figure 8 - *HLS* basic configuration.

HLS consists in 3 major parts:

- Server component

  Responsible for taking input streams of media and encoding them digitally, encapsulating them in a format suitable for delivery and preparing the encapsulated media for distribution.

  The encoder delivers the encoded media in an *MPEG-2 Transport Stream* over the local network. The streaming will then be segmented.

  The component responsible for streaming segmentation reads the *Transport Stream* from the local network and divides it into a series of small media files of equal duration. It also creates an index file containing references to the individual media files. This index is used to track the availability and location of the media files. Media segments are saved as *.ts* files (*MPEG-2 Transport Stream* files) and the index files are saved as *.M3U8* playlists.

- Distribution component

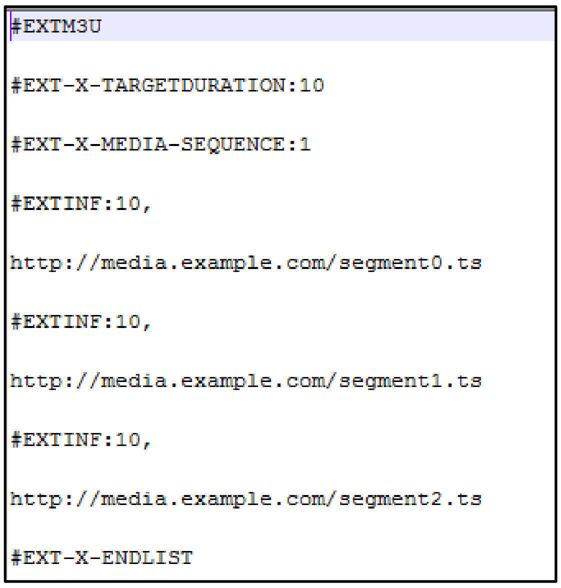
  Responsible for accepting client requests and delivering prepared media and associated resources to the client. It consists on a web server or a web caching system that delivers the media files and index files to the client over *HTTP*.

- Client software

  Responsible for determining the appropriate media to request, downloading the content of the requests, and reassembling them, so that the media can be presented to the user in a continuous streaming session.

  It begins by fetching the index file, based on a *URL* identifying the stream, which specifies the location of the available media files, decryption keys and any alternate streams available. For the selected stream, the client downloads each available media file in sequence.

This process continues until the client encounters the *#EXT-X-ENDLIST* tag in the index file. If this tag isn't present, the index file is part of an ongoing broadcast. In this case the client loads a new version of the index file periodically.



```
#EXTM3U

#EXT-X-TARGETDURATION:10

#EXT-X-MEDIA-SEQUENCE:1

#EXTINF:10,

http://media.example.com/segment0.ts

#EXTINF:10,

http://media.example.com/segment1.ts

#EXTINF:10,

http://media.example.com/segment2.ts

#EXT-X-ENDLIST
```

Figure 9 - Index file example.

2.2.2.2. Microsoft – Smooth Streaming

Smooth Streaming [7] is an *IIS Media Services* extension that enables adaptive streaming of media to clients over *HTTP*. The format specification is based on the *MP4* file specification and standardized by *Microsoft*. It is divided in 2 parts:

- Disk file format – Defines the structure of the contiguous file on disk, enabling better file management.

  The basic unit of an *MP4* file is called a "box", which contains both data and metadata.

  The file starts with file-level metadata ('*moov*') that generically describes the file, but the bulk of the payload is actually contained in the fragment boxes that also carry more accurate fragment level metadata ('*moof*') and media data ('*mdat*'). Closing the file is an '*mfra*' index box that allows easy and accurate seeking within the file.

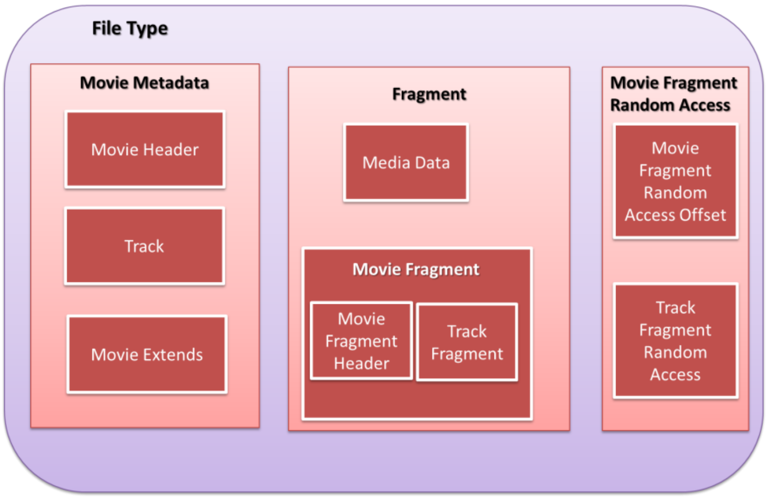


Figure 10 - *Smooth Streaming* File Format.

- Wire format – Defines the structure of the chunks that are send by *IIS* to the client.

  When a player client requests a video time slice from the *IIS Web server*, the server seeks to the appropriate starting fragment in the *MP4* file and then lifts the fragment out of the file and sends it over the wire to the client. This technique greatly enhances the efficiency of the *IIS Web server* because it doesn't induce any re-muxing or rewriting overhead.

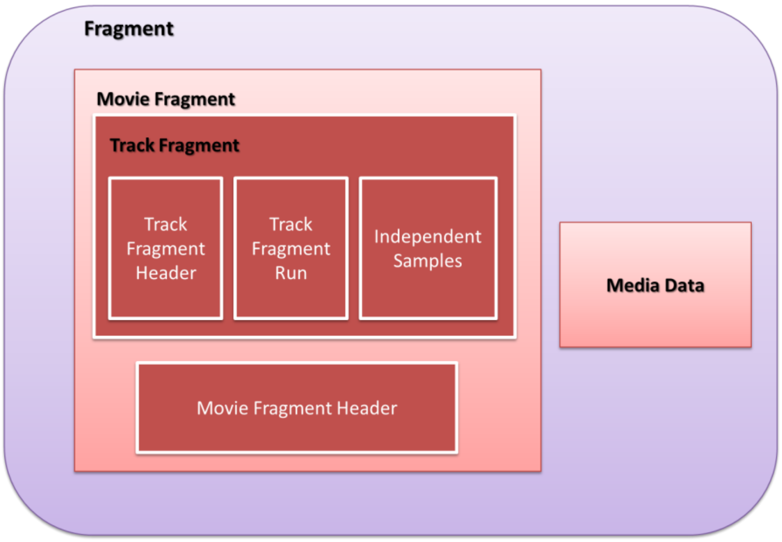


Figure 11 - *Smooth Streaming* Wire Format.

A typical Smooth Streaming media presentation consists of the following files:

- *MP4* files containing video/audio

- Server manifest file – Describes the relationships between the media tracks, bit rates and files on disk. Only used by the *IIS Server*, not by clients.
- Client manifest file – Describes the available streaming to the client: the codecs used, bit rates encoded, video resolutions, markers, captions, etc. It is the first file delivered to the client.

The first thing a multimedia player client requests from the server is the client manifest file, which tells it which codecs were used to compress the content (to initialize the correct decoder and build the playback pipeline), which bit rates and resolutions are available and a list of then available chunks.

After receiving the client request, *IIS Smooth Streaming* looks up the quality level (bit rate) in the corresponding server manifest file and maps it to a *MP4* file containing video/audio on the disk. It then figures out which fragment box corresponds to the requested start time offset, extracts and sends it over the wire to the client as a standalone file.

The server plays no part in the bit rate switching process. The client-side code looks at chunk's download times, buffer fullness, rendered frame rates and decides when to request higher or lower bit rates from the server.

2.2.2.3. Adobe - HDS

Abode Flash Player 10.1 software introduces support for *HTTP Dynamic Streaming* [8], enabling an adaptive-bitrate, protected streaming experience with common *HTTP* servers, catching devices and networks, using a standard *MP4* fragment format. *HTTP Dynamic Streaming* supports both live and on-demand media content that adjusts to the viewer connection speed and processing power, using standard *HTTP* protocol infrastructures that can meet the demand for *HD* content on a massive scale.

MP4 fragment-compliant files and manifest files are placed on an *HTTP* server that is responsible for receiving fragment requests over *HTTP* and returning the appropriate fragment from the file.
These fragments are downloaded and rendered by the *Flash Player* client. As the streaming is played, *ActionScript* within *Flash Player* monitors the client's bandwidth and playback experience, switching requests to the appropriate bit rate file fragments, improving playback performance.

At the start of the streaming session, the media player downloads the manifest file that provides all the information needed to play back the media, including fragment format, available bit rates, *Flash Access* licence server location and metadata information.

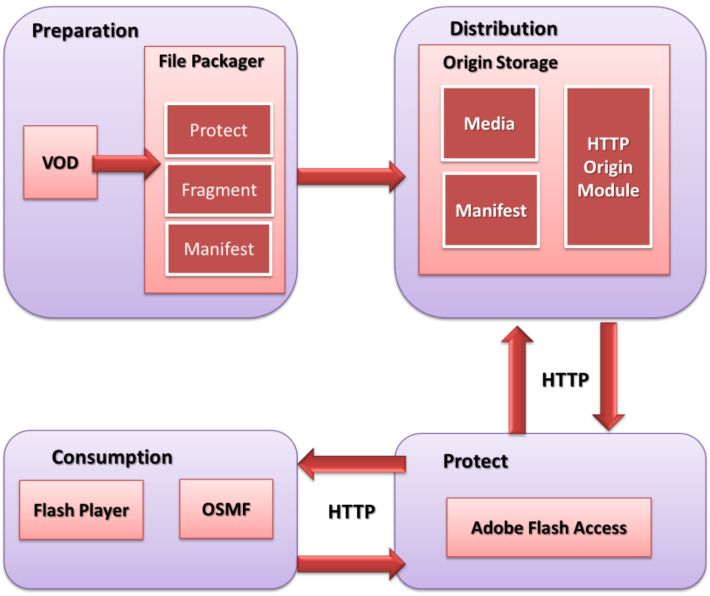


Figure 12 - *HTTP Dynamic Streaming.*

2.2.2.4. MPEG-DASH

MPEG-DASH is a technology developed by *MPEG* that enables high quality media content streaming over the Internet, using the *HTTP* protocol. This is the most recent adaptive stream protocol, putting together several features from the other related technologies (*HLS*, *HDS* and *Smooth Streaming*). *DASH* was published as a standard *(ISO/IEC 23009-1*) in April 2012.

The work around *DASH* is almost complete, it is expected his full release until later 2013. This protocol is the one used in this project for streaming multimedia content, so check Chapter 4 for a detailed characterization of *MPEG-DASH*.

2.3. Conclusions

For much of the past decade, it was quite difficult to do video streaming to a mobile device. Wide bandwidth variability, unfavorable firewall configurations and lack of network infrastructure support all created major roadblocks to streaming. Early, more traditional streaming protocols, designed for small packet networks, were anything but firewall friendly. Although *HTTP* progressive download was developed partially to get audio and video streams past throw firewalls, it still didn’t offer true streaming capabilities.

Adaptive Streaming gives a better experience to the user compared with to traditional streaming, because it can adjust the media to network conditions. Adaptive streaming is also much

more flexible, because it works over *HTTP*, so it doesn't have firewall or proxies issues and doesn't need special media servers, like traditional streaming does.
On the other hand, content preparation on adaptive streaming can be sometimes complex, adding storage costs and the bandwidth usage may not be as efficient as traditional streaming.

Despite being a technology developed by *Apple Inc*, *Android* (version 3+ or higher) natively supports *HLS*. *Android 3.0 Honeycomb* was release on beginning of 2011 and it was the first *Android* update for a tablet, so it had to support some adaptive streaming technology, because media consumption on tablets is a big deal and at the time it doesn't existed a standard technology.

In fact, *HLS* was proposed to the *IETF* for becoming standard in 2009, but no additional steps appear to been taken towards that *IETF* standardization.

In this dissertation, it was decided to use the *MPEG-DASH* protocol to provide the streaming functionality. This choice was based on the fact that *DASH* is an open standard and is based on very well-known technologies (namely, *MPEG-2* and *XML*), besides the fact of being the most recent alternative for supporting in a seamless way the adaptive streaming of audiovisual content in *IP* networks. In fact, being one of the goals the development of a solution as universal as possible, to be able to run on any type of mobile device, or at the expense of a small amount of effort for porting the solution to different platforms, DASH seems to be the most adequate alternative. The fact that it is to become an international standard approved by major standardization bodies, places *DASH* in a very good position to be widely adopted by the industry on the years to come.

Chapter 3

Android API

In recent years it has been observed a huge increment in mobile technologies usage such as smartphones, PDAs, tablets and notebooks.

Smartphones are the most complete technology for the user, because it puts together on the same terminal, capacity for network connection, data synchronization with a computer, contact phonebook and several sensors and input/output components.

Aside from its regular use as a phone, there are several additional applications with a large variety of uses. These applications can be developed by regular users, becoming available to everyone to download and use it easily.

There are a large variety of smartphones, with differences between them on the hardware and software level. On the software level there are several operating systems: *Google – Android, Apple – IOS, Microsoft – Windows Mobile, Nokia – Symbian, Black Berry – RIM*, etc.

On the hardware level, there are a large variety of manufactures that use the same operating system in their terminal. Therefore, is typical to find, for example, *Android* operating system in terminals with very distinct characteristics between them, such as dimensions, image and sound quality, memory availability and camera power.

So, it urges the need to do media content adaptation. Identifying the characteristics of each smartphone, before media content adaptation, is very important for this process to be possible, providing this way a better experience for the user.

3.1. Android Development

Android [11] is an operating system based on *Linux* with a *Java* programming interface. It was created by the *Open Handset Alliance*, led by *Google*. It's the most popular operating system, used by a huge variety of terminals and, for being open source, it's easy to develop applications for this platform.

3.1.1. Developing Tools

3.1.1.1. Java

Java [9] is a class-based and object-oriented computer programming language. It was released on 1995 by *Sun Microsystems*.

Java applications are typically compiled to bytecode that can run on any *JVM* regardless of computer architecture (in *Android*, it is used *Dalvik* virtual machine). The applications are compiled in bytecode that is interpreted by the *JVM* and transformed to binary code on the application execution.

To use *Java*, end-users need *JRE*, which contains the parts of the *Java SE* platform (latest release) required to run Java programs and *JDK*, which is intended for software developers, including development tools such as the *Java* compiler, *Javadoc*, *Jar* and debugger. End-users can also use a *Web* browser for *Java Applets*.

Java Applets are programs that are embedded in other applications, typically in a web page. *Java Servlets* are applets that run on the server side. These *Java EE* components generate responses to clients' requests.

In this project it was used *JRE 7* and *JDK 1.7.0_15*.

3.1.1.2. Eclipse

Eclipse [10] is a multi-language software development environment. It is used to develop Applications in *Java* and, by means of various plug-ins, other programming languages. The initial codebase originated from *IBM VisualAge*. In 2004 the *Eclipse Foundation* was created (non-profit organization).

Eclipse is one of the most used *IDE* for *Java* based applications development and the most used for *Android* applications development. For this, it uses the *ADT* plugin.

In this project, *Eclipse Classic 4.2.2* was used to develop the *Android* application and *Eclipse IDE for Java EE Developers* was used to develop the servlet on the server side.

Android development can be done using *Netbeans* as well as the command line, using a text editor to edit Java and *XML* files, than use tools such as *Java Development Kit* and *Apache Ant* to create, build and debug *Android* applications.

3.1.1.3. Android SDK

SDK is used to create new applications for the *Android* operating system. It includes a set of development tools, including debugger, libraries, emulator, documentation, sample codes and tutorials for the different *Android API* levels.

3.1.2. Installation and Configuration

Before starting with *Android* development, the reader needs to be sure he has the development environment set up. This project was developed on a *Windows* 7 (64-bit) computer, so the explanation will focus on how to install and configure the developing tools on *Windows* computers. For other platforms, the reader should check the *Android Developers* page for more information.

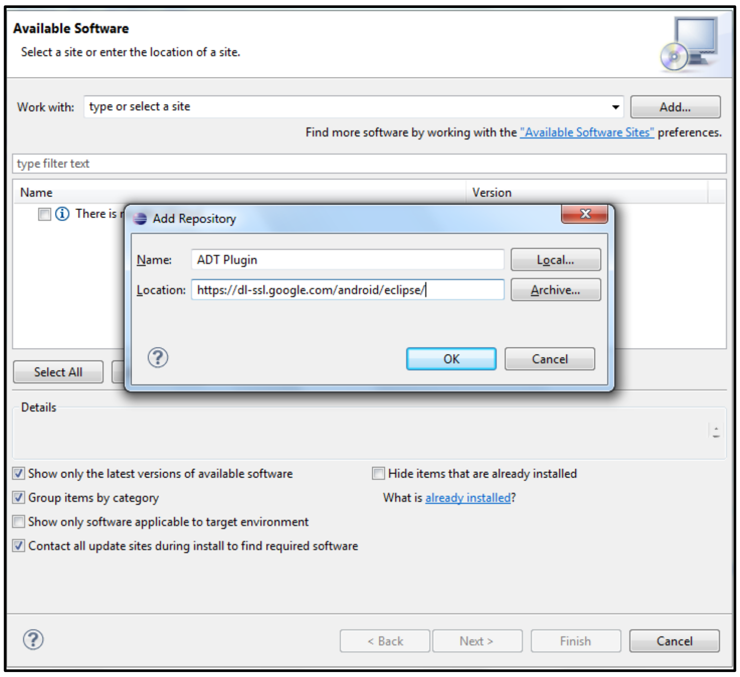
1) Download the *ADT Bundle*, if the reader wants to quickly start developing apps. It includes the essential *Android SDK* components and a version of the *Eclipse IDE* with built-in *ADT*. If the reader is using already *Eclipse* or want to use another *IDE*, he may choose to download and install first the *Android SDK Tools* and then the *ADT Plugin* for *Eclipse*.
   - ➔ *ADT Bundle* - http://developer.android.com/sdk/index.html, click on the top right corner button "*Download the SDK – ADT Bundle for Windows*".
   - ➔ *Android SDK Tools* - http://developer.android.com/sdk/index.html, open "*USE AN EXISTING IDE*" on the bottom of the page and click in "*Download the SDK Tools for Windows*".

   The reader can also download these tools for other platforms. Browse to http://developer.android.com/sdk/index.html, open "*DOWNLOAD FOR OTHER PLATFORMS*" and choose the packages suitable for his platform.

2) If the reader downloaded the *ADT Bundle*, simply unpack the *ZIP* file, browse to *Eclipse* directory and launch *Eclipse*. If the reader downloaded the *SDK Tools* only, to use with an existing *IDE*, he needs to install *SDK Tools*. When setting up the *ADT Plugin* the reader will need the name and location in which he installed the *SDK* on the system. So he must make note of these paths for later use.

3) After *SDK Tools* installation, the reader needs to install and configure *ADT Plugin* for *Eclipse*.

   - ➔ Start *Eclipse*, then select **Help**>**Install New Software** and click **Add**.
   - ➔ Enter "ADT Plugin" for the name, "https://dl-ssl.google.com/android/eclipse/" for the Location and select **OK**. The reader can also use "http" instead of "https" is he has trouble acquiring the plugin.
   - ➔ Check the *Developer Tools* and click **Next** to see a list of the tools to be downloaded.
   - ➔ Read and accept the license agreements and click **Finish**. If the reader gets a security warning, ignore it by clicking **OK**.


Available Software

Select a site or enter the location of a site.

Work with: type or select a site

Find more software by working with the "Available Software Sites" preferences.

type filter text

Name	Version
☐ There is r	

Add Repository

Name: ADT Plugin

Location: https://dl-ssl.google.com/android/eclipse/

Details

☑ Show only the latest versions of available software

☑ Group items by category

☐ Show only software applicable to target environment

☑ Contact all update sites during install to find required software

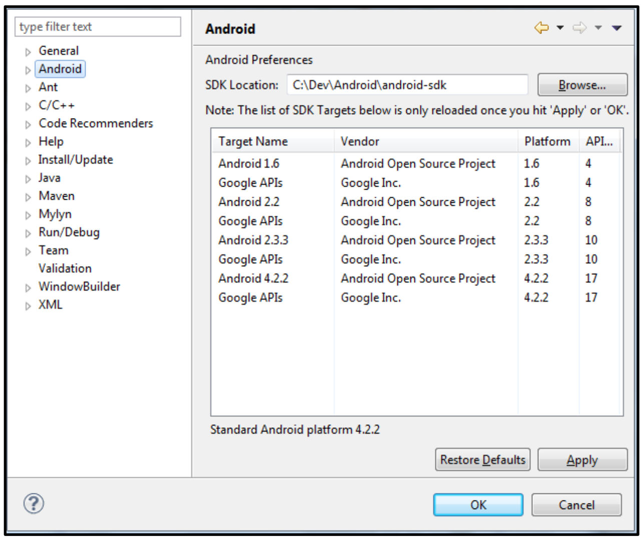
☐ Hide items that are already installed

What is already installed?

Figure 13 - Add *ADT* Plugin on *Eclipse*.

- ➔ When the installation completes, restart *Eclipse*.
- ➔ Once *Eclipse* restarts select **Use existing SDKs** in the "*Welcome to Android Development*" window and specify the location of the *Android SDK* directory.

  The reader can also go to *Window> Preferences> Android*, browse *SDK* location and click **OK**.


type filter text

General
Android
Ant
C/C++
Code Recommenders
Help
Install/Update
Java
Maven
Mylyn
Run/Debug
Team
Validation
WindowBuilder
XML

Android

Android Preferences

SDK Location: C:\Dev\Android\android-sdk

Note: The list of SDK Targets below is only reloaded once you hit 'Apply' or 'OK'.

Target Name	Vendor	Platform	API...
Android 1.6	Android Open Source Project	1.6	4
Google APIs	Google Inc.	1.6	4
Android 2.2	Android Open Source Project	2.2	8
Google APIs	Google Inc.	2.2	8
Android 2.3.3	Android Open Source Project	2.3.3	10
Google APIs	Google Inc.	2.3.3	10
Android 4.2.2	Android Open Source Project	4.2.2	17
Google APIs	Google Inc.	4.2.2	17

Standard Android platform 4.2.2

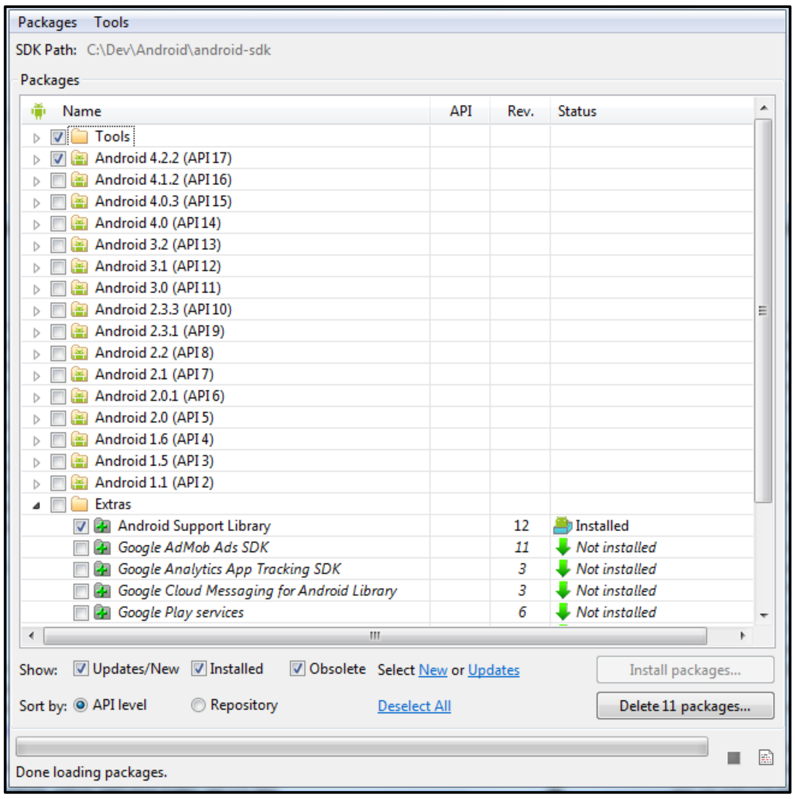
Figure 14 - Browse *SDK* location on *Eclipse*.

4) Download the latest *SDK* tools and platforms using the *SDK Manager*.

- ➔ Launch the *SDK Manager* (.exe file at the root of the *Android SDK* directory).

  The reader can also go to *Window> Android SDK Manager*.

➔ The reader must choose and install the components. He should install the latest Tools packages, the *Android* Support Library (on the Extras folder) and the *Android API* version he wants to support his future app (usually the latest one).



Packages Tools

SDK Path: C:\Dev\Android\android-sdk

Packages

	Name	API	Rev.	Status
✓	Tools			
✓	Android 4.2.2 (API 17)			
	Android 4.1.2 (API 16)			
	Android 4.0.3 (API 15)			
	Android 4.0 (API 14)			
	Android 3.2 (API 13)			
	Android 3.1 (API 12)			
	Android 3.0 (API 11)			
	Android 2.3.3 (API 10)			
	Android 2.3.1 (API 9)			
	Android 2.2 (API 8)			
	Android 2.1 (API 7)			
	Android 2.0.1 (API 6)			
	Android 2.0 (API 5)			
	Android 1.6 (API 4)			
	Android 1.5 (API 3)			
	Android 1.1 (API 2)			
	Extras			
✓	Android Support Library		12	Installed
	Google AdMob Ads SDK		*11*	*Not installed*
	Google Analytics App Tracking SDK		*3*	*Not installed*
	Google Cloud Messaging for Android Library		*3*	*Not installed*
	Google Play services		*6*	*Not installed*

Show: [✓ Updates/New] [✓ Installed] [✓ Obsolete] Select New or Updates

Sort by: [● API level] ○ Repository Deselect All

Install packages...

Delete 11 packages...

Done loading packages.

Figure 15 – Download the latest *SDK* tools and platforms support using the *SDK Manager*.

3.1.3. Android Project General Structure

3.1.3.1. Project Creation

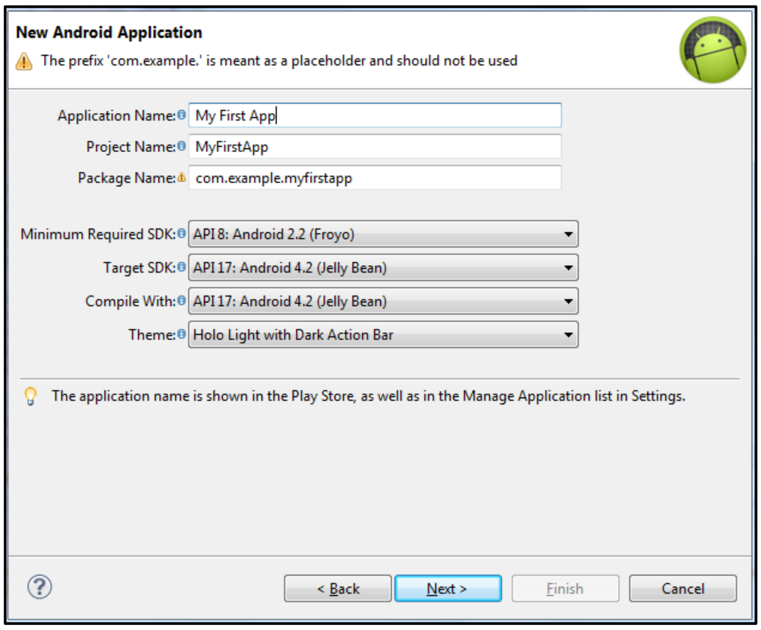
A new *Android* project can be created from *Eclipse* with *ADT* or from the command line. Since *Eclipse* was used to develop this project, the explanation will focus on how to do it in *Eclipse*. For more information about using the command line, please check the *Android Developers* page - http://developer.android.com/tools/projects/projects-cmdline.html.

➔ Select **File**> **New**> **Project**

➔ Select **Android**> **Android Application Project**, and click **Next**

➔ Enter the basic settings for the project: *Application Name* (title of the application launcher icon); *Project Name* (title of the folder where the project is created); *Package Name* (title of the initial package structure of the application); *Minimum Required SDK* (lowest version of the *Android* platform that the application supports); *Target SDK* (highest version of *Android* with which the application will work), and click **Next**.

➔ The reader can uncheck "*Create custom launcher icon*" and "*Create activity*" to finish the project creation or he can check the boxes to specify a launcher icon or create a default main activity.


New Android Application

The prefix 'com.example.' is meant as a placeholder and should not be used

Application Name: My First App

Project Name: MyFirstApp

Package Name: com.example.myfirstapp

Minimum Required SDK: API 8: Android 2.2 (Froyo)

Target SDK: API 17: Android 4.2 (Jelly Bean)

Compile With: API 17: Android 4.2 (Jelly Bean)

Theme: Holo Light with Dark Action Bar

The application name is shown in the Play Store, as well as in the Manage Application list in Settings.

Figure 16 - Project Creation on *Eclipse*.

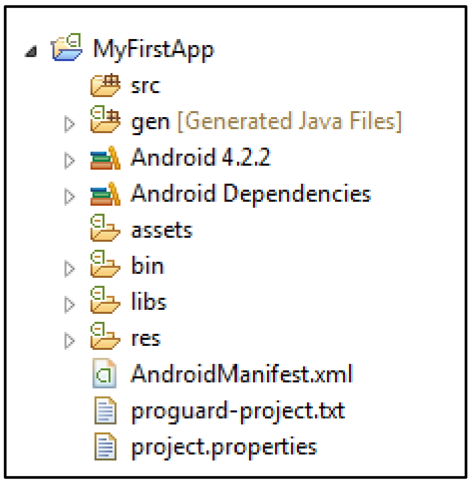
3.1.3.2. Project General Structure

Android projects get build into an *.apk* file that is installed into a device. It contains the application source code and resource files, generated by default or created by the developer. *Android* project comprise the following directories and files:

- *src/* - All source code files, such as *.java* go here.
- *bin/* - Output directory of the build. The final *.apk* file and other compiled resources go here.
- *jni/* - Contains native code sources developed using *NDK*.
- *gen/* - Contains the java files generated by *ADT*, such as *R.java*.
- *assets/* - Used to store raw assets files to be accessed by the application.
- *res/* - Contains application resources, such as drawable files, layout files and string values.
- *libs/* - Contains private libraries.
- *AndroidManifest.xml* – Control file that describes the nature of the application and each of its components. Is describes certain qualities about activities, services, intent receivers and content providers. It also describes what permissions are requested, what

external libraries are needed, what device features are required and what *API* levels are supported.

- *project.properties* – This file contains project settings, such as the build target.



```
◢ MyFirstApp
    src
  ▷ gen [Generated Java Files]
  ▷ Android 4.2.2
  ▷ Android Dependencies
    assets
  ▷ bin
  ▷ libs
  ▷ res
    AndroidManifest.xml
    proguard-project.txt
    project.properties
```

Figure 17 - *Android* project general structure.

3.1.4. API Levels

API level is an integer value that uniquely identifies the framework *API* revision offered by a version of the *Android* platform. The framework *API* is used by applications to interact with the underlying *Android* system.

Each successive version of the *Android* platform can include updates to the *Android* application framework *API* that it delivers. Because most changes in the *API* are additive and introduce new or replacement functionality, all other *API* parts from earlier revisions are carried forward without modification.

Table 1 - *Android API* levels.

Platform Version	*Codename*	*API Level*
1.6	*Donut*	4
2.1	*Eclair*	7
2.2	*Froyo*	8
2.3 - 2.3.2	*Gingerbread*	9
2.3.3 - 2.3.7	*Gingerbread*	10
3.2	*Honeycomb*	13
4.0.3 - 4.0.4	*Ice Cream Sandwich*	15
4.1.x	*Jelly Bean*	16
4.2.x	*Jelly Bean*	17

So, if the application developed is targeting the most recent *API* level, some functions it implements may not work on older *API* level devices. On the other hand, the most recent *API* level device will most likely run successfully functions of the old one.

Choosing an *API* level for an application development should take at least two things into account:

1. Number of devices that can actually support the application. At the date the most common *Android* version is *Gingerbread* [29], so if the target of the project is Ice *Cream Sandwich, Gingerbread* devices won't run it. This will be holding the application to reach more devices.
2. Choosing a lower *API* level may support more devices, but gain less functionality for the app. The user may also work harder to achieve features he could've easily gained if a higher *API* level was chosen.

3.1.5. Vitamio

Vitamio [12] is an open multimedia framework/library for *Android*, with hardware acceleration and renderer. It uses several open source projects, such as *FFmpeg* and *UniversalCharDet*. *FFmpeg* provides a software decoder and demuxers for output. *Vitamio LGPLv2.1* is licensed under the *FFmpeg* code. *UniversalCharDet* is the encoding detector library of *Mozilla*. *Vitamio MPL* is licensed under the *UniversalCharDet* to detect the encoding of subtitle texts.

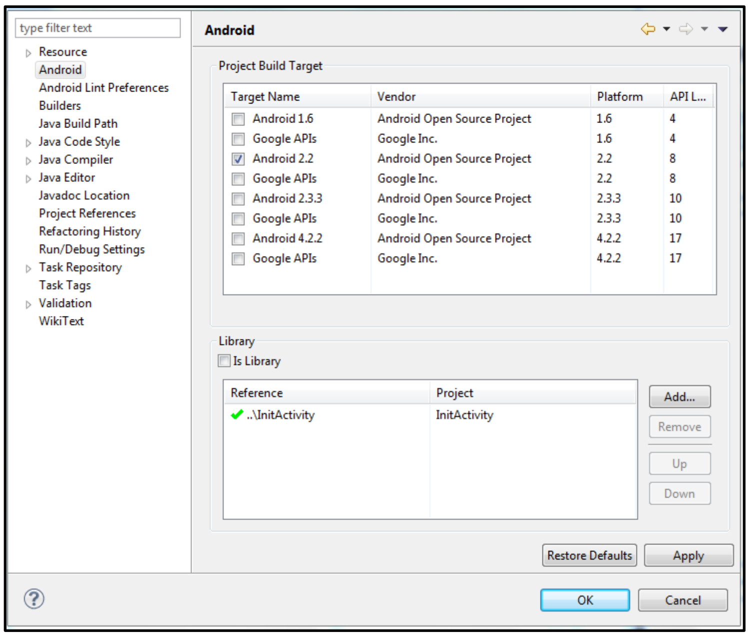
Vitamio supports *RTSP*, *RTMP*, *HTTP* progressive streaming and *HLS* streaming protocols. It can play almost all popular video formats, such as *MP4*, *MKV*, *M4V*, *MOV*, *FLV*, *AVI*, *RMVB*, *TS*, *3GP*, etc. *Vitamio* also supports the display of subtitle, external and embedded.

As far as *Vitamio's* device support, he supports two kinds of *ARM CPU*: *ARMv6* and *ARMv7*, although several problems were documented in various devices. The *ARM* architecture describes computer processors designed and licensed by *ARM Holdings*. According to the company, this *CPU* architecture is used in 95% of smartphones.

This library is used in this project because it is a powerful tool to deal with multimedia content, but it's not a requirement for *Android* development to handle multimedia content. The application can use for this the *Android* default tools.

It only supports playback and buffering of the file, but downloading the media content, exporting or processing buffered files is not supported. So, this library was used to decode the downloaded media content and play it.

- ➔ Download *VitamioBundle* (this is the *Android* library) - https://github.com/yixia/VitamioBundle
- ➔ Download *VitamioDemo (*demo project*)*, if the reader wants to test the library or he doesn't have his own project to bundle it - https://github.com/yixia/VitamioDemo
- ➔ UnZip and save *VitamioBundle* in the workspace folder of *Eclipse*, where it should be also the reader's project.
- ➔ Open *Eclipse*, *File> Import> Existing Projects into Workspace* and click **Next**.
- ➔ Browse the workspace directory and import the library (and the project, if wasn't already imported).
- ➔ On *Eclipse*, right click on the *VitamioBundle >Properties> Android*, check at the bottom *"Is Library"* and click **OK**.
- ➔ Right click on the project, *>Properties> Android*, add at the bottom the *Vitamio's* library and click **OK**.


type filter text
Resource
Android
Android Lint Preferences
Builders
Java Build Path
Java Code Style
Java Compiler
Java Editor
Javadoc Location
Project References
Refactoring History
Run/Debug Settings
Task Repository
Task Tags
Validation
WikiText

Android

Project Build Target

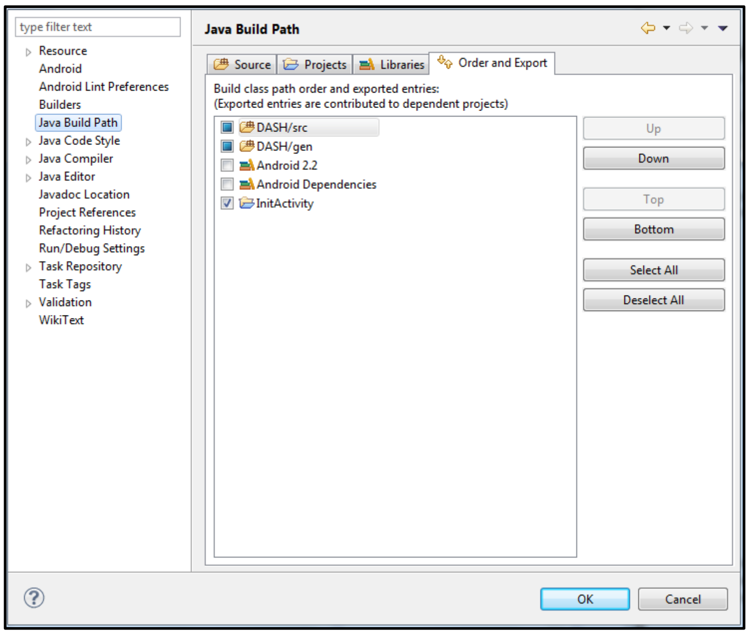
	Target Name	Vendor	Platform	API L...
☐	Android 1.6	Android Open Source Project	1.6	4
☐	Google APIs	Google Inc.	1.6	4
☑	Android 2.2	Android Open Source Project	2.2	8
☐	Google APIs	Google Inc.	2.2	8
☐	Android 2.3.3	Android Open Source Project	2.3.3	10
☐	Google APIs	Google Inc.	2.3.3	10
☐	Android 4.2.2	Android Open Source Project	4.2.2	17
☐	Google APIs	Google Inc.	4.2.2	17

Library
☐ Is Library

Reference	Project
✓ ..\InitActivity	InitActivity

Figure 18 - Configure *Vitamio* on *Android* project.

- ➔ If the *Vitamio* library is not presented on the last step, the reader needs to build the path manually. Right click on the project, *>Build Path> Configure Build Path...> Order and Export*, check the library's project and click **OK**. Now try to do the step before this one.



Build class path order and exported entries:
(Exported entries are contributed to dependent projects)

- ■ DASH/src
- ■ DASH/gen
- ☐ Android 2.2
- ☐ Android Dependencies
- ☑ InitActivity

Figure 19 - Manually build *Vitamio's* path on *Android* project.

Now that the library is bundle to the project, his use is the same as the default *Android* commands to handle video and audio reproduction, with no need for separated plugins download. The library components must be declared in the manifest file of the project.

3.2. Android Sensors

Android sensors [11] are used to measure a physical quantity, process the data input and do something with it. There are many components in a smartphone used for processing external environment data such as microphone for audio capture, camera for picture capture and *GPS* for location related data input.

3.2.1. Motion Sensors

Motion sensors measure acceleration forces and rotational forces along three axes. *Android* supports:

- Accelerometer
- Gravity sensor
- Gyroscope
- Linear accelerometer
- Rotation vector sensor

These sensors can indicate if the device is moving, using the accelerometer. If the user is moving, this probably means that he could be moving outdoors and using a *3G* network or he could be moving indoors and the *Wi-Fi* signal strength will not be linear.

With the measures made with these sensors, the user can only assume that the network connectivity won't be linear and stable as it should be.

3.2.2. Environment Sensors

Environment sensors measure various environmental parameters, such as ambient air temperature and pressure, illumination, and humidity. *Android* supports:

- Light sensor
- Pressure sensor
- Temperature sensor
- Humidity sensor

The light sensor can be very useful to measure lighting conditions. With these measurements, it can be possible to calibrate the screen brightness, giving a better experience to the user and it will do a better management of the device's battery lifetime.

3.2.3. Position Sensors

Position sensors measure the physical position of a device. *Android* supports:

- Orientation sensor
- Proximity sensor
- Geomagnetic Field sensor

This kind of sensors can be useful in many ways.

The orientation sensor measures if the device is on the vertical position or in landscape. With this information, the video resolution can be adapt to the current orientation of the device. Normally it is used the landscape orientation for better viewing experience on a video playback, but the user can reproduce the video with vertical orientation, urging the need for resolution adaptation.

The proximity sensor is used, as default, to turn off the screen during a phone call. When the user makes a phone call, this sensor will measure the proximity between the phone and the user's face/ear.

In video playback, the user can trigger some video features by simply putting a finger on top of the sensor or approximate it to his face. A good feature would be zooming the video or change subtitle's size.

The Geomagnetic Field sensor can be used the same way as *GPS*, it returns the location of the user. With this information, it can be identified if the user is in public places, which probably means the use of unsafe public *Wi-Fi* networks or even places with poor Internet connection.

3.2.4. Microphone

The microphone is used for audio input. With the audio input it could be measure the environment noise level and in result, the volume of the playback could be adapted in function of that noise measurement.

A headphone input must be checked first, because if the user has headphones, probably the environment noise won't matter and the volume doesn't need to be changed.

3.2.5. Camera

The camera is used for image capture. The image, with the proper image processing and analyses, can be used to identify a large variety of objects. It could identify faces, so if there is more than one person in the room, there may be some concerns about privacy issues.

It can identify the lighting conditions, if doesn't exist the light sensor on the device.

In can even identify cultural icons such as buildings, monuments or statues. So maybe the user is watching a video near by the *Eiffel Tower* and the audio idiom can change to French.

Chapter 4

MPEG-DASH

MPEG - DASH [13-18] is a technology developed by *MPEG* that enables high quality media content streaming over the Internet, using *HTTP* protocol. The following figure illustrates a simple streaming scenario between an *HTTP Server* and a *DASH Client*.

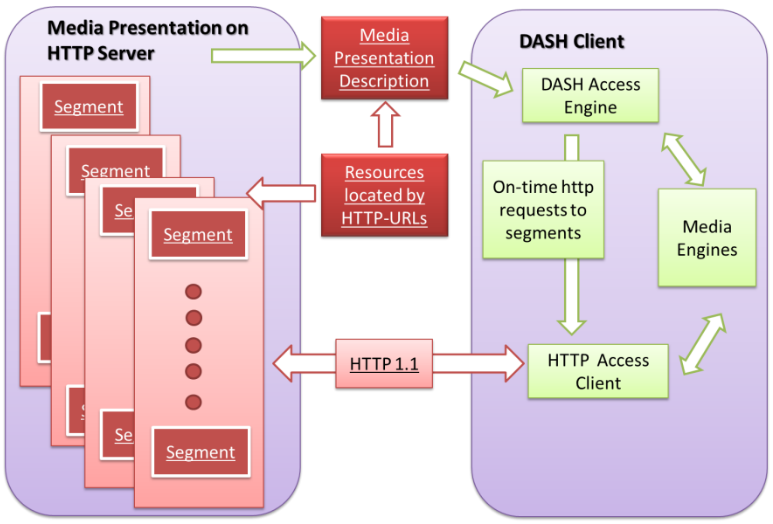

Media Presentation on HTTP Server
Segment
Segment
Segment
Segment
Media Presentation Description
Resources located by HTTP-URLs
HTTP 1.1
DASH Client
DASH Access Engine
On-time http requests to segments
Media Engines
HTTP Access Client

Figure 20 - General *MPEG-DASH* system architecture.

4.1. DASH General Operation

In Figure 20, the underlined components are the only ones specified by *DASH*. These components are responsible for generate content and handle metadata, such as resources location and format.

The content on the server is divided in two categories: segments and metadata file. The segments contain the actual multimedia. They can be separated files or a single file with multiple sub-segments.

The *DASH* client first obtains the metadata file and parses it to learn about the content characteristics, such as media types, media-content availability, resolutions, minimum and maximum bandwidth, accessibility features, media-components location on the network, etc. Using this information, the *DASH Client* selects the appropriate encode alternative and starts streaming the content by fetching the segments using *HTTP GET* requests.

Highlighted features:

- Supports on demand and live adaptive streaming
- Supports both *ISO BMFF (MP4)* and *MPEG-2 TS*.
- Codec agnostic.
- Efficient and easy use of existing *CDNs*, proxies, caches, *NATs* and firewalls.
- Control of entire streaming session by the client.
- Support of seamless switching of tracks.
- Segments with variable durations.
- Segment alignment indication to simplify switching and avoiding overlapping fragments.
- Supports all *DRM* techniques specified in *ISO/IEC 23001-7: Common Encryption*.
- Allow sub-segments, retrieved by *HTTP* byte range requests.
- Supports trick modes for seeking, fast forwards and rewind.

4.2. MPD

4.2.1. Profiles

MPEG-DASH specifies six different *Profiles*. A *Profile* is a set of restrictions on the offered *Media Presentation* related to encoding of the segments and the source of the stream session (it could be live or on demand). These *Profiles* can be used as permission for *DASH* clients that only implement the features required by the profile to process the *Media Presentation*.

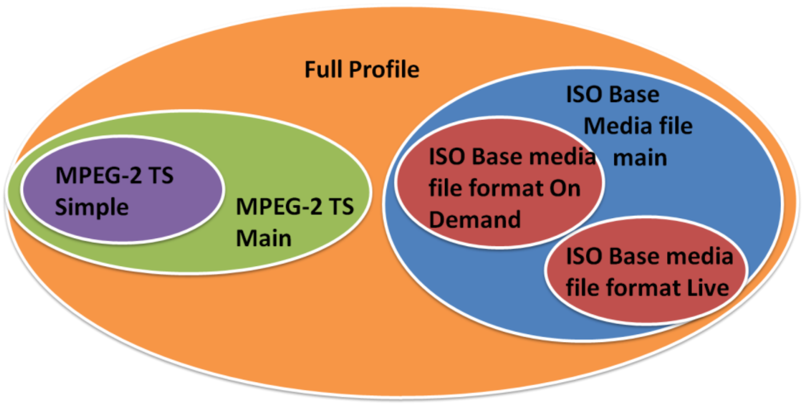
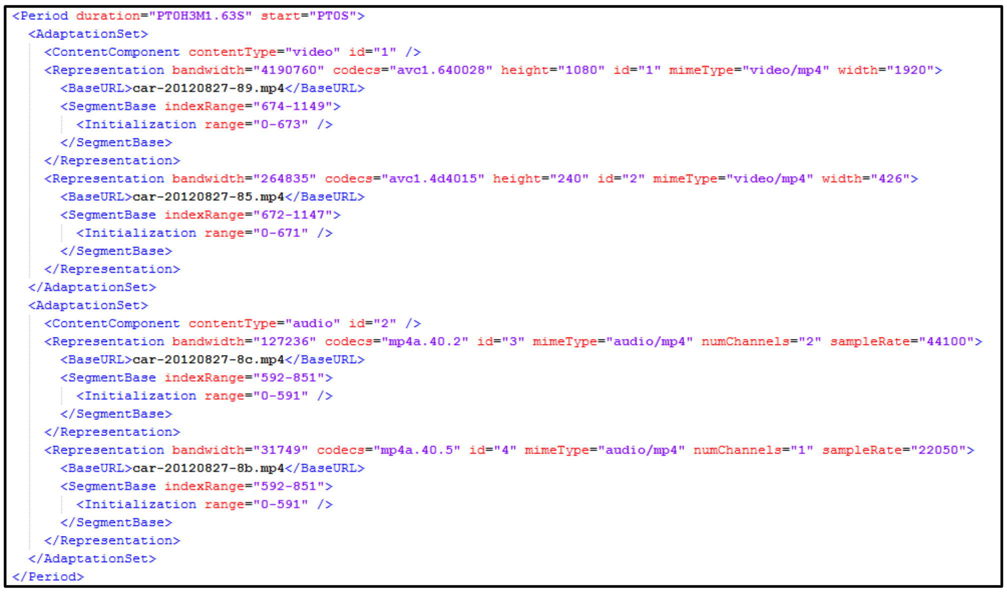

Full Profile
MPEG-2 TS Simple
MPEG-2 TS Main
ISO Base Media file main
ISO Base media file format On Demand
ISO Base media file format Live

Figure 21 - *MPEG-DASH Profiles* defined in *ISO/IEC 23009*.

4.2.1.1. ISO Base media file format Profiles

There are three *Profiles* related to *ISO* based encoded files:

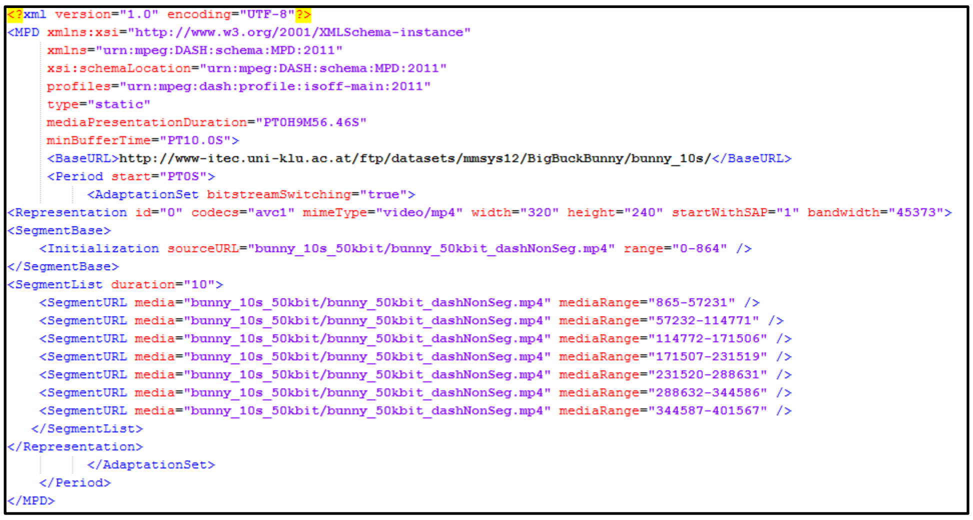
1. *ISO Base media file format On Demand* – intended to provide basic support for on demand content, supporting large *VoD* libraries with minimum amount of content management.



```xml
<Period duration="PT0H3M1.63S" start="PT0S">
  <AdaptationSet>
    <ContentComponent contentType="video" id="1" />
    <Representation bandwidth="4190760" codecs="avc1.640028" height="1080" id="1" mimeType="video/mp4" width="1920">
      <BaseURL>car-20120827-89.mp4</BaseURL>
      <SegmentBase indexRange="674-1149">
        <Initialization range="0-673" />
      </SegmentBase>
    </Representation>
    <Representation bandwidth="264835" codecs="avc1.4d4015" height="240" id="2" mimeType="video/mp4" width="426">
      <BaseURL>car-20120827-85.mp4</BaseURL>
      <SegmentBase indexRange="672-1147">
        <Initialization range="0-671" />
      </SegmentBase>
    </Representation>
  </AdaptationSet>
  <AdaptationSet>
    <ContentComponent contentType="audio" id="2" />
    <Representation bandwidth="127236" codecs="mp4a.40.2" id="3" mimeType="audio/mp4" numChannels="2" sampleRate="44100">
      <BaseURL>car-20120827-8c.mp4</BaseURL>
      <SegmentBase indexRange="592-851">
        <Initialization range="0-591" />
      </SegmentBase>
    </Representation>
    <Representation bandwidth="31749" codecs="mp4a.40.5" id="4" mimeType="audio/mp4" numChannels="1" sampleRate="22050">
      <BaseURL>car-20120827-8b.mp4</BaseURL>
      <SegmentBase indexRange="592-851">
        <Initialization range="0-591" />
      </SegmentBase>
    </Representation>
  </AdaptationSet>
</Period>
```

Figure 22 – *MPD* example from an *ISO Base media file format On Demand Profile*.

2. *ISO Base media file format Live* – optimized for live encoding and low latency delivery of Segments consisting of a single movie fragment of *ISO* file format with relatively short duration. *Smooth Streaming* content can be integrated with this profile.
3. *ISO Base media file format Main* – support for on demand and live content.



```xml
<?xml version="1.0" encoding="UTF-8"?>
<MPD xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns="urn:mpeg:DASH:schema:MPD:2011"
     xsi:schemaLocation="urn:mpeg:DASH:schema:MPD:2011"
     profiles="urn:mpeg:dash:profile:isoff-main:2011"
     type="static"
     mediaPresentationDuration="PT0H9M56.46S"
     minBufferTime="PT10.0S">
     <BaseURL>http://www-itec.uni-klu.ac.at/ftp/datasets/mmsys12/BigBuckBunny/bunny_10s/</BaseURL>
     <Period start="PT0S">
          <AdaptationSet bitstreamSwitching="true">
<Representation id="0" codecs="avc1" mimeType="video/mp4" width="320" height="240" startWithSAP="1" bandwidth="45373">
<SegmentBase>
    <Initialization sourceURL="bunny_10s_50kbit/bunny_50kbit_dashNonSeg.mp4" range="0-864" />
</SegmentBase>
<SegmentList duration="10">
    <SegmentURL media="bunny_10s_50kbit/bunny_50kbit_dashNonSeg.mp4" mediaRange="865-57231" />
    <SegmentURL media="bunny_10s_50kbit/bunny_50kbit_dashNonSeg.mp4" mediaRange="57232-114771" />
    <SegmentURL media="bunny_10s_50kbit/bunny_50kbit_dashNonSeg.mp4" mediaRange="114772-171506" />
    <SegmentURL media="bunny_10s_50kbit/bunny_50kbit_dashNonSeg.mp4" mediaRange="171507-231519" />
    <SegmentURL media="bunny_10s_50kbit/bunny_50kbit_dashNonSeg.mp4" mediaRange="231520-288631" />
    <SegmentURL media="bunny_10s_50kbit/bunny_50kbit_dashNonSeg.mp4" mediaRange="288632-344586" />
    <SegmentURL media="bunny_10s_50kbit/bunny_50kbit_dashNonSeg.mp4" mediaRange="344587-401567" />
   </SegmentList>
</Representation>
          </AdaptationSet>
    </Period>
</MPD>
```

Figure 23 - *MPD* example from *ISO Base media file format Main Profile*.

4.2.1.2. MPEG-2 TS Profiles

There are two *Profiles* related to *MPEG-2TS* encoded files:

1. *Main* - Imposes little constrains on the Media Segment format for *MPEG-2 Transport Stream* content. *HLS* content can be integrated with this profile, because both use *MPEG-2 TS* as video content format.

2. *Simple* – A subset of the main profile. Poses more restrictions on the encoding and multiplexing in order to allow simple implementation of seamless switching.

The previous *Profiles* simplify the content organization of the metadata file according to the encoding of the segments. There is also the *Full Profile* that is the most generalized *Profile*, supporting *ISO Base* or *MPEG-2 TS* media file format.

4.2.2. Hierarchical Model

MPD is a *XML* document compliant to *DASH,* used to save metadata information of the media content. It describes different bit rate representations of the media resources, enabling dynamic adaptive streaming over *HTTP*. The *MPD* file contains fully qualified *URLs* to the segments and it expresses the relationship between these and the corresponding representation.

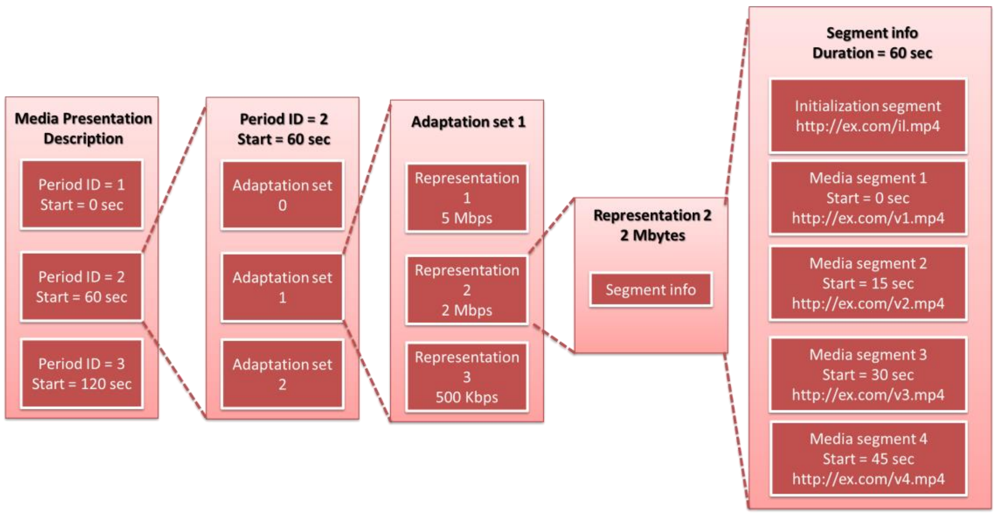

Media Presentation Description
Period ID = 1 Start = 0 sec
Period ID = 2 Start = 60 sec
Period ID = 3 Start = 120 sec
Period ID = 2 Start = 60 sec
Adaptation set 0
Adaptation set 1
Adaptation set 2
Adaptation set 1
Representation 1 5 Mbps
Representation 2 2 Mbps
Representation 3 500 Kbps
Representation 2 2 Mbytes
Segment info
Segment info Duration = 60 sec
Initialization segment http://ex.com/il.mp4
Media segment 1 Start = 0 sec http://ex.com/v1.mp4
Media segment 2 Start = 15 sec http://ex.com/v2.mp4
Media segment 3 Start = 30 sec http://ex.com/v3.mp4
Media segment 4 Start = 45 sec http://ex.com/v4.mp4

Figure 24 - *MPD* Hierarchical Data Model.

Periods represent temporal intervals of the media content. Typically is used only one *Period* with the entire media file duration, but it can be used more for better information organization, if the movie's length is high. Each *Period* consists of one or more *Adaptation Sets*.

Adaptation Sets provide the information about one or more *Content Component* and its various encoded alternatives. These are used to organize the information about different media component types, such as video and audio. There could be only one *Adaptation Set* about video and others about different audio tracks for that movie (used to watch the movie in more than one idiom). The simplest form is only one *Adaptation Set* for video and audio together. Each *Adaptation Set* includes multiple *Representations*.

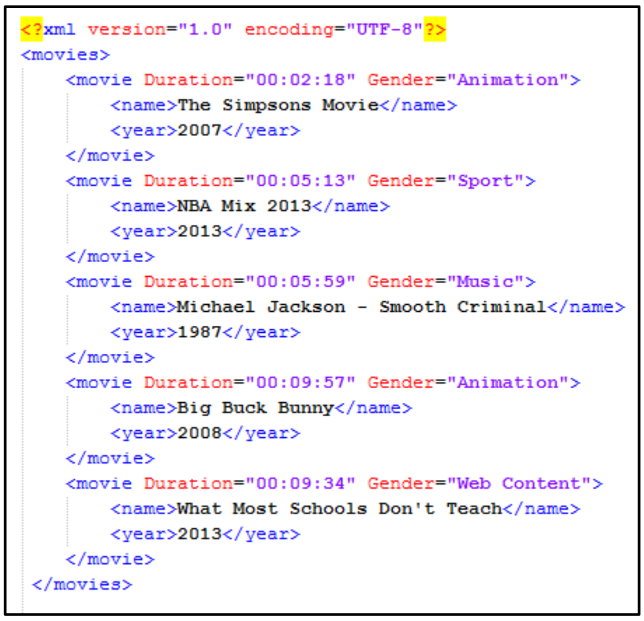
Representations are encoded alternatives of the same media component, varying from other *Representations* by bit rate, resolution, media component type, language, etc. For the video adaptation in the playback to be possible, the same media content has to be encoded with different parameters, such as bit rate for different video and audio quality and resolution for different screen sizes. Each *Representation* is identified by an identification number and includes a list of consecutive *Segments* of the media content.

Segments are the media stream chunks, result from splitting the media file and organize the content information in small units. *Segments* are referenced by *HTTP URL* that addresses their location on the server, to be accessed using *HTTP GET* requests.

4.2.3. File Structure

4.2.3.1. XML General Structure

XML is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. It is used to store, transport and display data in a very simple way.



```
<?xml version="1.0" encoding="UTF-8"?>
<movies>
    <movie Duration="00:02:18" Gender="Animation">
        <name>The Simpsons Movie</name>
        <year>2007</year>
    </movie>
    <movie Duration="00:05:13" Gender="Sport">
        <name>NBA Mix 2013</name>
        <year>2013</year>
    </movie>
    <movie Duration="00:05:59" Gender="Music">
        <name>Michael Jackson - Smooth Criminal</name>
        <year>1987</year>
    </movie>
    <movie Duration="00:09:57" Gender="Animation">
        <name>Big Buck Bunny</name>
        <year>2008</year>
    </movie>
    <movie Duration="00:09:34" Gender="Web Content">
        <name>What Most Schools Don't Teach</name>
        <year>2013</year>
    </movie>
 </movies>
```

Figure 25 - *XML* simple example.

XML documents are text files, consisting entirely of characters. These characters may be divided into markup and content. Strings that constitute a markup are generally identified by the characters '<' and '>', in the beginning and in the end. Strings that constitute content don't.

The most common markup construct in a *XML* file is identified as a *Tag*. This component that begins with '<' and ends with '>' exists in three forms: *Start-tags*, *End-tags* and *Empty-element tags*.

Elements are components that start with a start-tag and end with a matching end-tag, or consist only of an empty-element tag. All the characters between the tags consist on the element's content, which may include other elements called *Child Elements*.

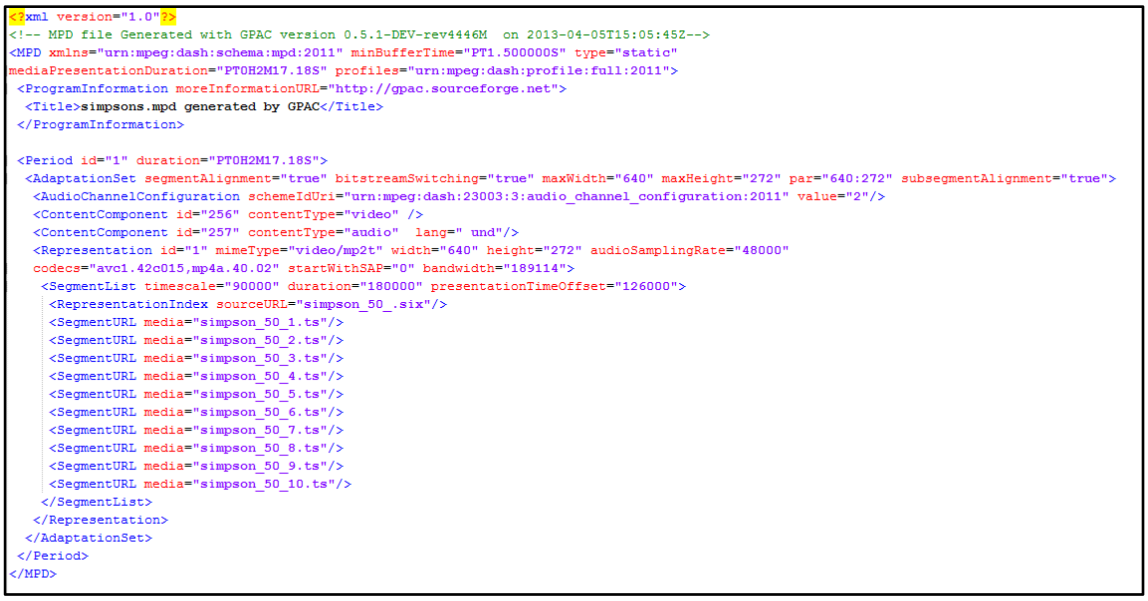
Attributes are markup constructs consisting of a name/value pair that exists within a start-tag or an empty-element tag.

The *XML* example above describes a list of movies. <*movies*> is a Start-tag and </*movies*> is his matching End-tag. Together they constitute the "*movies*" *Element*, which include a list of "*movie*" *Elements* and witch one includes two other *Elements*, identifying the name and year of production. The *Element's* content is the string between matching tags.

On the "*movie*" *Element* exists two attributes that characterizes the movie, namely the *Duration* and the *Gender*.

4.2.3.2. Media Presentation Description

As said before, *MPD* is the designation of the metadata file used by *DASH* specification. This file represents the metadata using *XML* language, according to the hierarchical model. The *MDP* structure may vary according to the Profile chosen for encoding. In this project was used the *Full Profile*, because it supports both media encoding types, so the explanation will focus on this profile structure. Nevertheless, for other profiles, the elements and attributes are very similar. From now on, *MPD* elements will be identified between '<>' and attributes will begin with '@'.



```xml
<?xml version="1.0"?>
<!-- MPD file Generated with GPAC version 0.5.1-DEV-rev4446M  on 2013-04-05T15:05:45Z-->
<MPD xmlns="urn:mpeg:dash:schema:mpd:2011" minBufferTime="PT1.500000S" type="static"
mediaPresentationDuration="PT0H2M17.18S" profiles="urn:mpeg:dash:profile:full:2011">
 <ProgramInformation moreInformationURL="http://gpac.sourceforge.net">
  <Title>simpsons.mpd generated by GPAC</Title>
 </ProgramInformation>

 <Period id="1" duration="PT0H2M17.18S">
  <AdaptationSet segmentAlignment="true" bitstreamSwitching="true" maxWidth="640" maxHeight="272" par="640:272" subsegmentAlignment="true">
   <AudioChannelConfiguration schemeIdUri="urn:mpeg:dash:23003:3:audio_channel_configuration:2011" value="2"/>
   <ContentComponent id="256" contentType="video" />
   <ContentComponent id="257" contentType="audio"  lang=" und"/>
   <Representation id="1" mimeType="video/mp2t" width="640" height="272" audioSamplingRate="48000"
   codecs="avc1.42c015,mp4a.40.02" startWithSAP="0" bandwidth="189114">
    <SegmentList timescale="90000" duration="180000" presentationTimeOffset="126000">
     <RepresentationIndex sourceURL="simpson_50_.six"/>
     <SegmentURL media="simpson_50_1.ts"/>
     <SegmentURL media="simpson_50_2.ts"/>
     <SegmentURL media="simpson_50_3.ts"/>
     <SegmentURL media="simpson_50_4.ts"/>
     <SegmentURL media="simpson_50_5.ts"/>
     <SegmentURL media="simpson_50_6.ts"/>
     <SegmentURL media="simpson_50_7.ts"/>
     <SegmentURL media="simpson_50_8.ts"/>
     <SegmentURL media="simpson_50_9.ts"/>
     <SegmentURL media="simpson_50_10.ts"/>
    </SegmentList>
   </Representation>
  </AdaptationSet>
 </Period>
</MPD>
```

Figure 26 - *MPD* example of *Full Profile* with segment's *URL*.

- <*MPD*> is the root element and stores general information such as profile identification with *@profiles*, the duration of the entire media presentation with *@mediaPresentationDuration* and if the *MPD* may be updated or not with *@type* ("*@type=dynamic*" if updated and "*@type=static*" if not). There is other attributes that may be present, if "*@type=dynamic*", such as *@availabilityStartTime* to specify the time that segments may become available and *@minimumUpdatePeriod* to control the frequency at which a client checks for updates.

- <*Period*> is a child element of <*MPD*>. It has attributes such as *@id* for identification and *@duration* to identify the length of that *Period*.

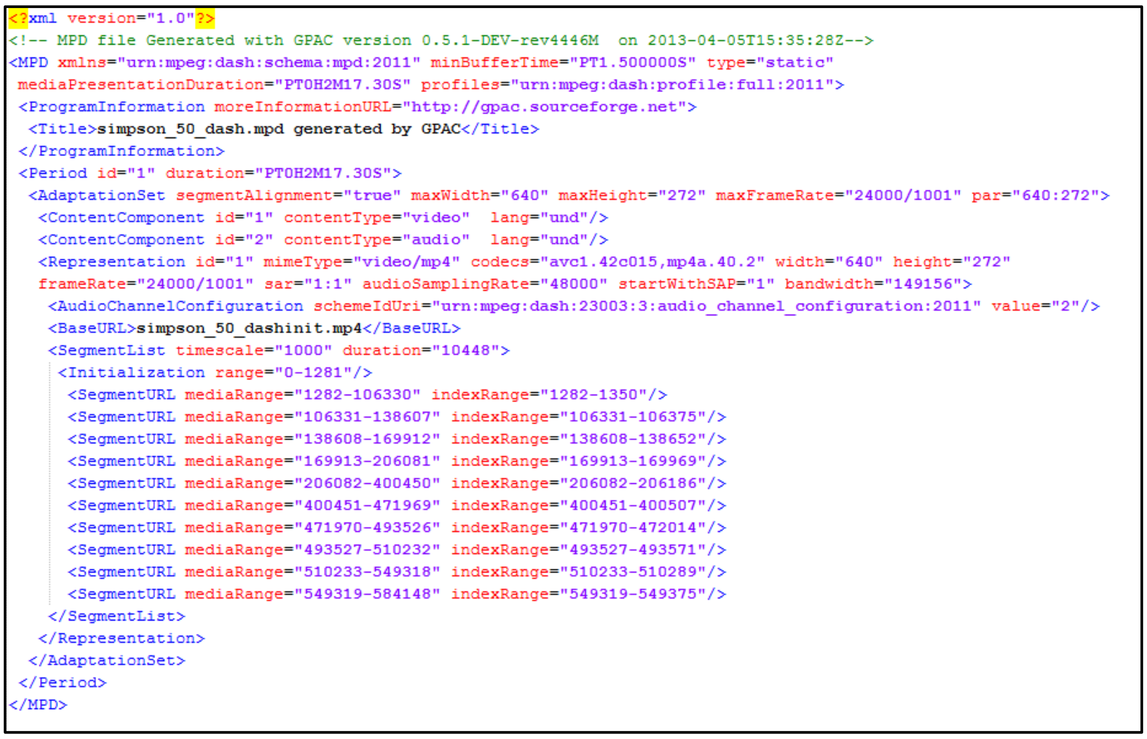
- <*AdaptationSet*> is a child element of <*Period*>. It is usually associated with it one or more <*ContentComponent*>, to identify the content type with *@contentType*. <*AdaptationSet*> has attributes such as *@segmentAlignment* and

@subsegmentAlignment, to allow non-overlapping decoding and presentation of segments from different *Representations*, *@bitstreamSwitching*, to allow concatenation of segments from different *Representations* results in conforming bitstream, *@maxWidth*, *@maxHeight*, *@par* and *@maxFrameRate* for video resolution information.

- <*Representation*> is a child element of <*AdaptationSet*>. It has a *@id* for identification, *@mimetype* and *@codecs* for content encoding information, *@width*, *@height*, *@frameRate* and *@sar* for video resolution information, *@audioSamplingRate* and *@bandwidth* for content bit rate information and *@startWidthSAP* to use the presentation time and position in segments at which random access and switching can occur.

- <*BaseURL*> to identify the *URL* of the index segment or, in other cases, it is used *@sourceURL* from the <*RepresentationIndex*>.

- <*SegmentList*> is the parent element of several <*SegmentURL*> that uses *@media* to identify the segment *URL* or *@mediaRange*. *@indexRange* is used to identify the intervals of bytes used for a partial *HTTP* request. If byte range was used, <*Initialization*> is used to specify the range of the index segment.



```xml
<?xml version="1.0"?>
<!-- MPD file Generated with GPAC version 0.5.1-DEV-rev4446M  on 2013-04-05T15:35:28Z-->
<MPD xmlns="urn:mpeg:dash:schema:mpd:2011" minBufferTime="PT1.500000S" type="static"
 mediaPresentationDuration="PT0H2M17.30S" profiles="urn:mpeg:dash:profile:full:2011">
 <ProgramInformation moreInformationURL="http://gpac.sourceforge.net">
  <Title>simpson_50_dash.mpd generated by GPAC</Title>
 </ProgramInformation>
 <Period id="1" duration="PT0H2M17.30S">
  <AdaptationSet segmentAlignment="true" maxWidth="640" maxHeight="272" maxFrameRate="24000/1001" par="640:272">
   <ContentComponent id="1" contentType="video"  lang="und"/>
   <ContentComponent id="2" contentType="audio"  lang="und"/>
   <Representation id="1" mimeType="video/mp4" codecs="avc1.42c015,mp4a.40.2" width="640" height="272"
   frameRate="24000/1001" sar="1:1" audioSamplingRate="48000" startWithSAP="1" bandwidth="149156">
    <AudioChannelConfiguration schemeIdUri="urn:mpeg:dash:23003:3:audio_channel_configuration:2011" value="2"/>
    <BaseURL>simpson_50_dashinit.mp4</BaseURL>
    <SegmentList timescale="1000" duration="10448">
     <Initialization range="0-1281"/>
      <SegmentURL mediaRange="1282-106330" indexRange="1282-1350"/>
      <SegmentURL mediaRange="106331-138607" indexRange="106331-106375"/>
      <SegmentURL mediaRange="138608-169912" indexRange="138608-138652"/>
      <SegmentURL mediaRange="169913-206081" indexRange="169913-169969"/>
      <SegmentURL mediaRange="206082-400450" indexRange="206082-206186"/>
      <SegmentURL mediaRange="400451-471969" indexRange="400451-400507"/>
      <SegmentURL mediaRange="471970-493526" indexRange="471970-472014"/>
      <SegmentURL mediaRange="493527-510232" indexRange="493527-493571"/>
      <SegmentURL mediaRange="510233-549318" indexRange="510233-510289"/>
      <SegmentURL mediaRange="549319-584148" indexRange="549319-549375"/>
    </SegmentList>
   </Representation>
  </AdaptationSet>
 </Period>
</MPD>
```

Figure 27 - *MPD* example of *Full Profile* with media ranges.

4.3. Media Content

Media content is digital information intended to be delivered to an end-user/audience. The content is compressed and stored in a server. Next it will be transported, decompressed and presented by the client. For this are used codecs and containers.

Codecs are responsible for compression and decompression of the media content. This is done to reduce the content size that will be transported in a transmission not 100% reliable and stored in limited memories. Later is decompressed and presented. The most popular codecs are *MPEG-1*, *MPEG-2*, *WMV* and *H.264*.

Containers are used for the content transportation (stream over Internet) and storage. Once the media data is compressed into suitable formats and reasonable sizes, it is packaged in containers to be transported to the end user. They are like "boxes" used to hold a variety of different codecs. The most common are *QuickTime*, *MP4*, *MPEG-2 TS*, *Matroska*, *WebM* and *Flash*.

4.3.1. Container Formats

MPEG–DASH uses *MPEG2-TS* and *MP4* as containers for the streaming of the media content. Because of *DASH* and the fact that it is an international standard, it is expected that video distributers will change the use from their current streaming specification to *DASH*.

MPEG2-TS is used in *HLS* and MP4 is used in *HDS* and *Smooth Streaming*. So, to make this change easier, *DASH* supports both container formats, allowing the reutilization of the encoded media content.

4.3.1.1. MPEG2-TS

MPEG2-TS is specified in *MPEG-2 Part1, Systems*. The content is packaged in a *PES*, the basic unit of data, structured with a header and a payload. The header contains information about the package, such as synchronization and packet identifier, and the payload contains the data content, such as video, audio content and metadata, which is not separated from the actual media data.

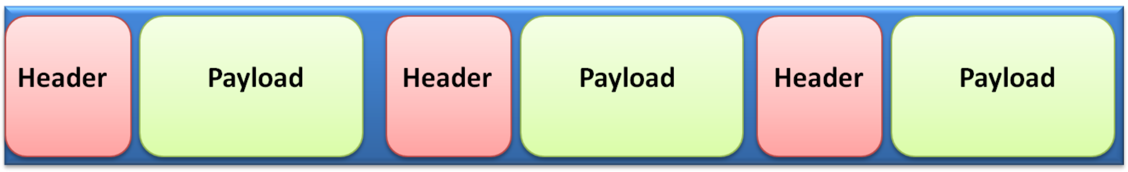


Figure 28 - Packetized Elementary Stream simple example.

4.3.1.2. MP4

MP4 is a standard specified as a part of *MPEG-4*, based on the *ISO* base media file format defined in *MPEG-4 Part 12* and *JPEG 2000 Part 12* which in turn was based on the *QuickTime* file format.

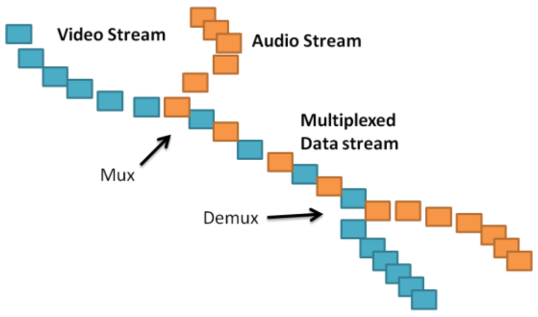


Figure 29 - Simple example of audio and video multiplexing.

The content is packaged in boxes or atoms, the basic unit of data. On contrary of the *MPEG-2 TS*, there are different kinds of boxes, distinguished by their content:

- *ftyp* – describes the file type and compatibility of the *MP4*. It is always present at the beginning.
- *moov* – contains all the descriptive and technical metadata, to allow the player to use appropriate codecs for the various elementary streams, identify them correctly, etc. The movie box itself contains a movie content header (*mvhd*) and description of elementary streams (*trak*).
- *mdat* – contains the multiplexed media data, raw audio and video information and timed-text elements that are decoded based on a movie box's information.

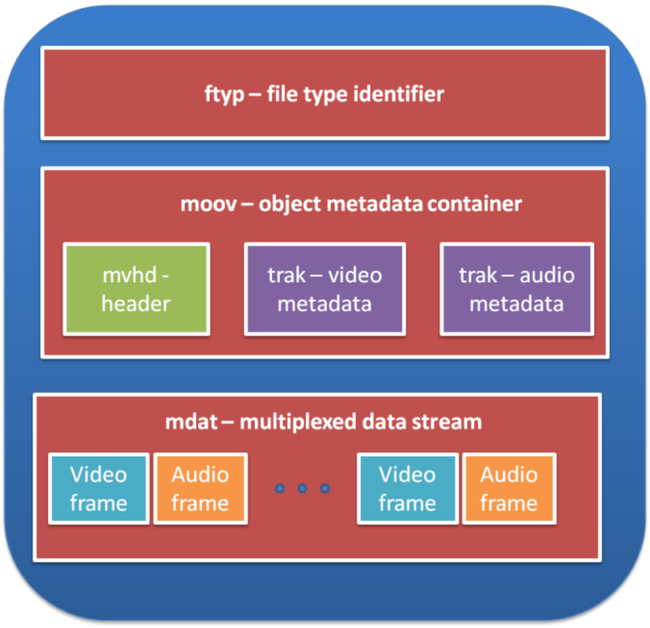


Figure 30 - Simple *MP4* file structure.

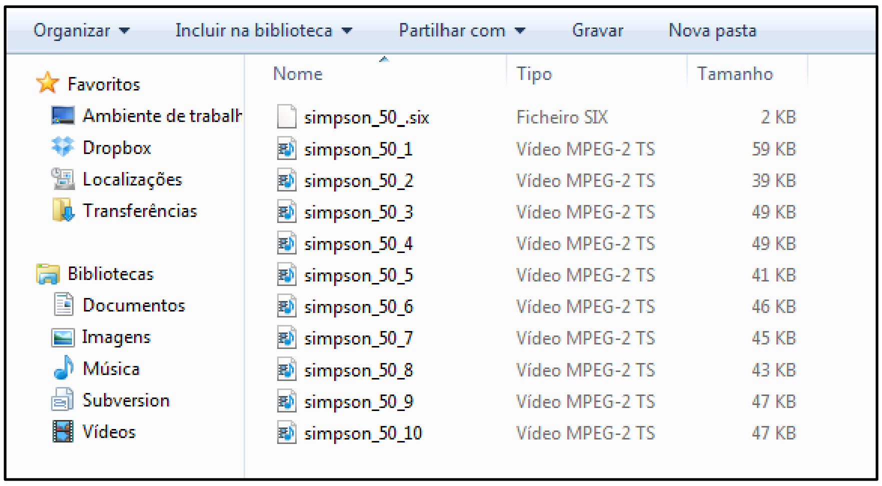
4.3.2. Segmentation

Before being requested, the media content must be prepared and stored on a server. As mentioned before, the media content must be segmented to be streamed, using *HTTP Adaptive Stream*.

The content is encoded into multiple quality levels and each one of those is segmented into equal sized segments. Now this segmentation can result into multiple separated segment files or into just one segment file with multiple sub-segments.

4.3.2.1. Separated Segments

In this method, the media content is segmented into separated segment files. These files are stored in a server waiting to be requested. The client needs to do an *HTTP* request to retrieve them, using the *URL* specified on the *MPD* of their location.

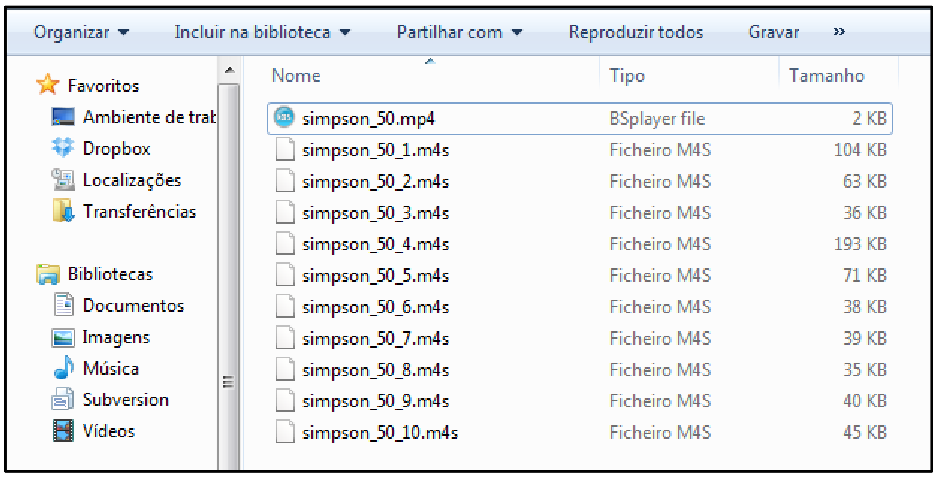


Nome	Tipo	Tamanho
simpson_50_.six	Ficheiro SIX	2 KB
simpson_50_1	Vídeo MPEG-2 TS	59 KB
simpson_50_2	Vídeo MPEG-2 TS	39 KB
simpson_50_3	Vídeo MPEG-2 TS	49 KB
simpson_50_4	Vídeo MPEG-2 TS	49 KB
simpson_50_5	Vídeo MPEG-2 TS	41 KB
simpson_50_6	Vídeo MPEG-2 TS	46 KB
simpson_50_7	Vídeo MPEG-2 TS	45 KB
simpson_50_8	Vídeo MPEG-2 TS	43 KB
simpson_50_9	Vídeo MPEG-2 TS	47 KB
simpson_50_10	Vídeo MPEG-2 TS	47 KB

Figure 31- *MPEG-2 TS* segment files' directory.

This is the layout of a *MPEG-2 TS* after being segmented into 10 separated segments. The *.six* file at the beginning is the *Initialization Segment*, used to initialize the media for playback. This file contains the media stream access points, marked frames within the streaming, allowing the segment switching on the playback. If no *Initialization Segment* is present, then each media segment is self-initialized.

In case the content is encoded as a *MP4* file, the *Initialization Segment* is an *.mp4* file instead of a *.six* and the media segments are *.m4s* instead of *.ts*, as shown in the next image.



Nome	Tipo	Tamanho
simpson_50.mp4	BSplayer file	2 KB
simpson_50_1.m4s	Ficheiro M4S	104 KB
simpson_50_2.m4s	Ficheiro M4S	63 KB
simpson_50_3.m4s	Ficheiro M4S	36 KB
simpson_50_4.m4s	Ficheiro M4S	193 KB
simpson_50_5.m4s	Ficheiro M4S	71 KB
simpson_50_6.m4s	Ficheiro M4S	38 KB
simpson_50_7.m4s	Ficheiro M4S	39 KB
simpson_50_8.m4s	Ficheiro M4S	35 KB
simpson_50_9.m4s	Ficheiro M4S	40 KB
simpson_50_10.m4s	Ficheiro M4S	45 KB

Figure 32 - *MP4* segments files' directory.

4.3.2.2. Single Segment

The media content may be segmented into a single segment file. This method is used to better organization of the segments in the segment folder. For example, if the file has 1 hour duration and is segmented into chucks of 1 second long, the end result will be 3600 segments.

With this method, the file is segmented internally into sub-segments. Each sub-segment is retrieved using partial *HTTP* requests. An *URL* specification will be replaced by a byte media range, constituted by the initial byte and the end byte, marking respectively the beginning and the end of the pretended sub-segment. Partial *HTTP* requests are only possible using *HTTP/1.1*.

The result is a single file, *.ts* if *MPEG2-TS* or *.mp4* if *MP4*. The file contains all the segments, including the Initialization Segment, which is identified the same way as the other by a byte range interval.

4.3.3. Reproduction

After being received by the *Dash Client*, the media content will be played. The *Dash Client* has to request the content, one segment at a time, and concatenate them into a file, that will be played.

First it will request the Initialization Segment and then the media segments. If the Initialization Segment isn't present at the beginning, the media player won't have the access points of the stream for the playback of the whole file to be possible.

The segments are downloaded and saved into a cache file. The cache will begin playback as soon as reaches a minimum amount of data, defined by the developer. The requested files should be of low quality to minimize the download time and start the playback as fast as possible.

After playback as started, the *Dash Client* will continue to request the following media segments of the stream and save them in the cache that is being played. So, if the playback reaches a certain frame that were not yet downloaded into the cache, the playback will stop. That's the reason that before requesting the segment, the *Dash Client* must find out what's the best quality to prevent crashing the playback.

The *Dash Client* has to run parallel processes to retrieve some context information (usually Internet characteristics and terminal capacities) and come up with the bandwidth availability at that time. Next, it will compare this value with the *@bandwidth* value of every <*Representation*> and choose one representation according to the following conditions: the *@bandwidth* value has to be lower that the available bandwidth and the highest of all lower values.

4.4. Media Coding

4.4.1. FFmpeg

FFmpeg [19] is a tool for handling multimedia data using various libraries. *FFmpeg* is developed under *GNU/Linux*, but can be compiled under most operating systems. This project is made of several components:

- *ffmpeg* is a command-line tool to convert one video file to another.
- *ffserver* is an *HTTP* and *RTSP* multimedia streaming server for live broadcasts.
- *ffplay* is a simple media player.
- *ffprove* is a command-line tool to show media information.
- *libswresample* is a library containing audio resampling routines.
- *libavcodec* is a library containing all the *FFmpeg* audio/video encoders and decoders.
- *libavformat* is a library containing demuxers and muxers for audio/video container formats.
- *libavutil* is a helper library containing routines common to different parts of *FFmpeg*.
- *libpostproc* is a library containing video post processing routines.
- *libswscale* is a library containing video image scaling and color space/pixel format conversion routines.
- *libavfilter* is the substitute for *vhook* which allows the video/audio to be modified or examined between the decoder and the encoder.

In this project, *FFmpeg* was used to encode audio and video with different bit rates and mux them in different containers (*MP4* and *MPEG-2 TS*).

4.4.1.1. Installation

FFmpeg is a multi-platform tool. *Windows 7 (64 bits)* was the chosen operating system to run it. So, the explanation about the installation and usage of *FFmpeg* will focus on the chosen platform. If the reader wants to know more about installation on other platforms, he should go to http://www.ffmpeg.org/download.html.

1. Download the *FFmpeg* *Windows* builds from http://ffmpeg.zeranoe.com/builds/.
2. Decompress the file, open the resulting folder, enter on bin directory and remember this file path.
3. Open the *Windows Command Line* and specify the previous file path.
4. Type *ffmpeg-h* to execute the *FFmpeg* help menu and confirm that *FFmpeg* is operational.

4.4.1.2. Usage

FFmpeg is a command line tool, so *Windows Command Line* were used to run some commands. Each command contains specific parameters for the action pretended to take place. In this case, audio and video of a *MP4* file were encoded with different bit rates. After that the output file is transcoded to other format, *MPEG2-TS* according to *DASH* specification.

1. Encode audio and video of a *MP4* file

➔ *ffmpeg -i input_file.mp4 -vcodec libx264 -vprofile baseline -preset slow -b:v 50k -maxrate 50k -bufsize 100k -vf scale=iw/3:-1 -acodec libvo_aacenc -b:a 96k output_file.mp4*

2. Transcode *MP4* file into *MPEG2-TS*

➔ *ffmpeg -i input_file.mp4 -vcodec copy -acodec copy -vbsf h264_mp4toannexb output_file.ts*

FFmpeg reads from an input file, specified by *-i* option, and writes to an output file, specified by a plain output filename. As a general rule, options are applied to the next specified file, so beware of the options order. Exceptions of this rule are the global options, which are specified always in the beginning (None used in this case).

- *-i input_file.mp4*: reads from a *MP4 input_file*.

- *-vcodec libx264*: specify *x264* as video codec. If the option copy is used, *FFmpeg* will not encode the video again, just use the same codec on file transcoding.
- *-vprofile baseline*: specify the *H.264* profile as baseline for highest compatibility with *Android* devices.
- *-present slow*: specify the speed of the encoding, in this case slow. A slower speed will provide a better compression and achieve a better file quality.
- *-b:v 50k*: specify the average video bit rate, in this case 50Kbps. This can be used in combination with *-maxrate* and *-bufsize* to prevent some of the swings and simulate a constant bit rate.
- *-vf scale=iw/3:-1*: specify the video resolution in scale with the original video. What this do is downsize by a factor of 3 the video width and automatically choose the right video height, maintaining the video proportions. It can be used a simpler option, typing *-s width x height*.
- *-acodec libvo_aacenc*: specify *aac* as audio codec. If the option copy is used, *FFmpeg* will not encode audio again, just use the same codec on file transcoding.
- *-b:a 96k*: specify the average audio bit rate, in this case 96Kbps.
- *-vbsf h264_mp4toannexb*: This filter allows a *MP4* file to be transcoded to a *MPEG2-TS* file.

4.4.2. MP4Box

MP4Box [20] is the multimedia packager available in *GPAC*, an open source multimedia framework for research and academic purposes, being developed by *Telecom ParisTech* as part of the research work of the multimedia group. *MP4Box* can be used for performing manipulations on several multimedia files, encoding/decoding presentation languages, performing encryption and attaching metadata to streaming sessions and preparation of *HTTP Adaptive Streaming* content.

In this project, *MP4Box* was used to generate content conformant to the *MPEG-DASH* specification.

4.4.2.1. Installation

GPAC can be installed on multiple platforms and his latest stable release is version 0.5.0. *Ubuntu 12.04* was used to install *GPAC*, so the explanation will focus on its installation and usage for this platform. If the reader wants to know how to do it for other platforms, he can check http://gpac.wp.mines-telecom.fr/downloads/.

Use *Ubuntu's Terminal* and type the following commands:

1. Get the source code:

➔ *sudo apt-get install subversion*

➔ *svn co https://gpac.svn.sourceforge.net/svnroot/gpac/trunk/gpac gpac*

2. Get the dependencies:

➔ *sudo apt-get install make pkg-config g++ zlib1g-dev firefox-dev libfreetype6-dev libjpeg62-dev libpng12-dev libopenjpeg-dev libmad0-dev libfaad-dev libogg-dev libvorbis-dev libtheora-dev liba52-0.7.4-dev libavcodec-dev libavformat-dev libavutil-dev libswscale-dev libxv-dev x11proto-video-dev libgl1-mesa-dev x11proto-gl-dev linux-sound-base libxvidcore-dev libssl-dev libjack-dev libasound2-dev libpulse-dev libsdl1.2-dev dvb-apps libavcodec-extra-53 libavdevice-dev libmozjs185-dev*

3. Compile:

➔ *cd gpac*

➔ *./configure*

➔ *Make*

➔ *sudo make install*

4.4.2.2. Usage

MP4Box is a command line tool, so the reader needs to open *Ubuntu's Terminal* and type the command for the specific action he wants to perform. According to the pretended action, the command can have different parameters. In this case, the content was generated in two forms:

1. Separated Segments:

➔ *MP4Box -dash <u>dur</u> -segment-name <u>name</u> <u>input_file</u>*

2. Single Segment:

➔ *MP4Box -dash <u>dur</u> -rap <u>input_file</u>*

- *-dash <u>dur</u>*: produce segments with duration *<u>dur</u>*, expressed in milliseconds.
- *-segment-name <u>name</u>*: generate each segment in a dedicate file named *<u>name</u>%d.ext*, being is a numeric counter and *ext* is the file's extension.
- *<u>input_file</u>*: name of the file to be segmented. It can be a *MP4* (*.mp4*) or *MPEG2-TS* file (*.ts*).

Note that the underlined text refers to the values of the parameters.

After the command is typed, the process will generate the segmented content and the *MPD* file associated. The *MPD* generated corresponds to the *Full Profile*. There are clearly differences in <*SegmentURL*> on both *MPDs*: one identifying the segment *URL* with @*media* and the other identifying the media range interval with @*mediaRange*.

4.5. Use Cases and Future Work

MPEG-DASH is still premature. Although it is on the break of his fully release, it's still a work in progress, putting together contributes of many institutions. There is not must information about it, especially about advanced topics.

It's not difficult to develop a dash client and generate his content, if the application sticks to basic functionalities. *MPEG-DASH* supports simple and advanced use cases. The simple ones can be gradually extended to become more complex and advanced. Some advanced topics are:

- Dynamic ad-insertion;
- Delivery of other multimedia content besides video and audio, such as *2D*, *3D*, animation, graphics, multiview, subtitles and text;
- Support of multiple languages and different audio configurations.

The next steps for *MPEG-DASH* are:

1. Complete the standardization work
   - Formal approval of all specifications
   - Conformance, interoperability and reference software
2. Towards deployments:
   - Generate guidelines, white papers, test content and software
   - Promotional efforts
   - Combine it with browsers, the web and *HTML5*
3. Migration Scenarios:
   - Most generated content and production equipment of the previous specifications of *HTTP Adaptive Stream* can be reused: *HLS* and *Smooth Streaming* content are suitable for *DASH MPEG-2 TS Main Profile* and *ISO BMFF Live Profile*, respectively.
   - Manifest files can be converted to *MPD* format: *XML* conversion from *m3u8* and *Smooth Streaming* manifests or deployment of other manifests in parallel as the *MPD*.

Chapter 5

System Architecture and Implementation

The project developed consists on a video streaming service for *Android* clients, using *MPEG-DASH* as protocol to generate and deliverer media content. The system is constituted by two different major modules, the Server side, which consists on a simple *HTTP* web server, and the Client side, which consists on an *Android* application.

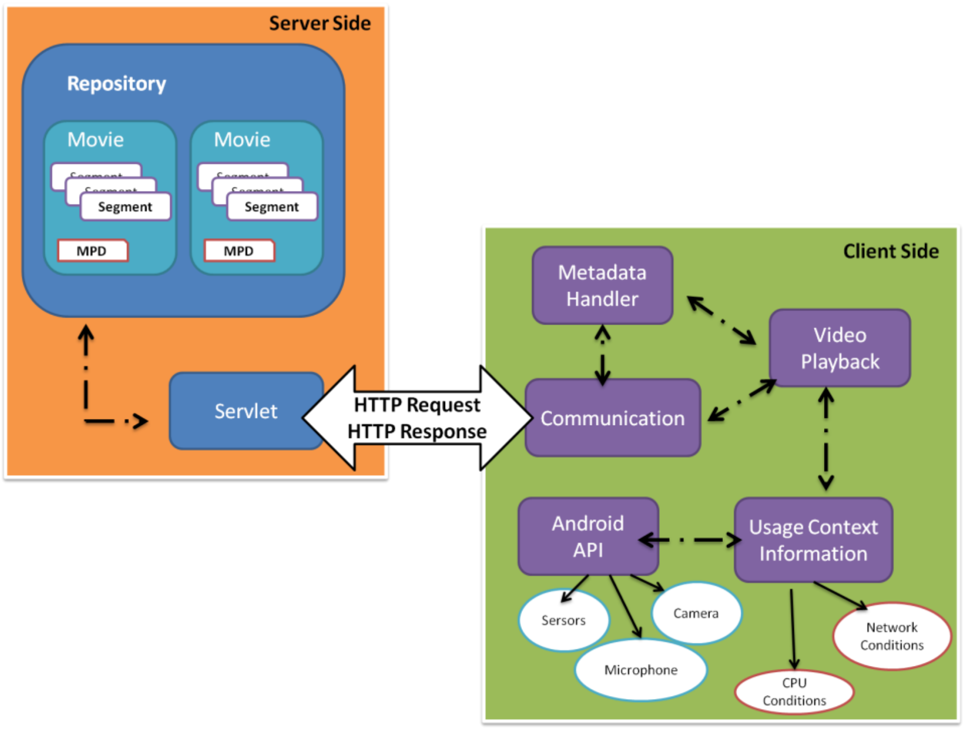


Figure 33 - *DASH Client* system architecture.

5.1. Server Side

As seen before, *MPEG-DASH* specifies how media content is generated and how metadata is handled. The media and metadata files are stored into a regular web server. His job is to receive requests, made by a client, for the media and metadata files and send them to be consumed. The media quality is chosen by the client before the request, so the server just has to receive requests and send the files requested.

Java Servlet API is a protocol by which a *Java* class may respond to requests. A *Servlet* is a Java class that conforms this *API*. The Servlet is used to communicate over any client-server protocol, often used with *HTTP* protocol. To deploy and run a *Servlet*, a *Web Container* must be used. This container is the component of a web server that interacts with the *Servlets*.

5.1.1. Technologies

In this project, the web server is constituted by a *Servlet*, which is managed by *Apache Tomcat*, an open source web server and a Servlet container developed by *Apache Software Foundation*. To work with *Tomcat*, a *Java* based *IDE* was used, namely *Eclipse*.

5.1.1.1. Eclipse & Apache Tomcat

Eclipse IDE for Java EE Developers was used to develop the servlet on the server side. On contrary of *Eclipse Classic 4.2.2*, the EE version includes web tools and therefore, is used to develop web applications.

Apache Tomcat [21] has different versions available for different versions of the *Servlet* specifications. Version 7 is the latest stable version, implementing *Servlet 3.0* specifications.

In this project was used *Apache Tomcat 7* with *Eclipse IDE for Java EE Developers*.

5.1.1.2. Installation

1. Go to http://www.eclipse.org/downloads/ and download *Eclipse IDE for Java EE Developers*.
2. Go to http://www.coreservlets.com/Apache-Tomcat-Tutorial/tomcat7-files/tomcat-7.0.34-preconfigured.zip and download *Apache Tomcat 7*.
3. Unzip both files.
4. Start *Eclipse*, R-click on the *Servers* tab at bottom (if the reader doesn't see the *Servers* tab, he can add the tab via **Window**-> **Show View**-> **Servers**) ->**New** -> **Server** -> **Apache** -> **Tomcat v7.0**.

5. On *Server runtime environment* click on **Add…** and browse to the folder where the reader unzipped *Tomcat* previously. He should now see "*Tomcat v7.0 Server at localhost*" listed under the *Servers* tab at the bottom.
6. R-click on "*Tomcat v7.0 Server at localhost*" and choose **Start** to run *Tomcat*. Open http://localhost/ and the reader will see the *Tomcat* welcome page.
7. If a 404 error message appears instead of the welcome page, it probably comes from *Tomcat*. This happens because *Eclipse* forgets to copy the default apps (Root, examples, etc) when it creates a *Tomcat* folder inside the *Eclipse* workspace. The reader needs to do this manually:

   -Navigate to the folder where he unzipped *Tomcat* -> *webapps* and copy the *ROOT* folder.

   - Navigate to the *Eclipse* workspace folder -> *.metadata* and search for "*wtpwebapps*"

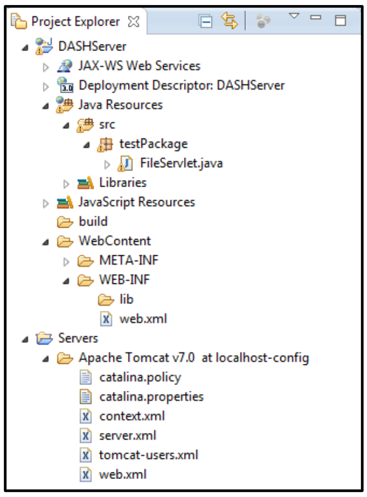
   - R-click on the *wtpwebapps* folder and past the *ROOT* folder. Say yes if asked to merge/replace folders/files.

   - Reload http://localhost/ to see the *Tomcat* welcome page.

5.1.2. Dynamic Web Project

Now that Tomcat is ready to go, the reader can use his machine to build *Web* pages dynamically. He has to create a *Dynamic Web Project* to run the java files to manage the server *Servlet*. He can also manage stored files he wants to be available online, in this case video and metadata files. To create the project the reader has to:

8. Go, on *Eclipse*, to **File**-> **New**-> **Other** -> **Web** -> **Dynamic Web Project**
9. Choose a *Project* name and click **Finish** (I this case it is *DASHServer*)
10. R-click on "*Tomcat v7.0 Server at localhost*" -> **Add and Remove**, add the created *Project* to configure it on the server and click **Finish**. If *Eclipse* asks to switch to the *Java EE Perspective* answer yes.



```
DASHServer
  JAX-WS Web Services
  Deployment Descriptor: DASHServer
  Java Resources
    src
      testPackage
        FileServlet.java
    Libraries
  JavaScript Resources
  build
  WebContent
    META-INF
    WEB-INF
      lib
      web.xml
Servers
  Apache Tomcat v7.0 at localhost-config
    catalina.policy
    catalina.properties
    context.xml
    server.xml
    tomcat-users.xml
    web.xml
```

Figure 34 - Dynamic Web Project general structure.

Locations:

- *Java Resources: src/testPackage/* – This folder contains the java files that control the *Servlet*. It is recommended to store these files inside a package and never to use the default package.

11. *R-Click on* ***Java Resources/src*** *->* ***Package***
12. Choose a name, in this case *testPackage*, and click ***Finish***

- *DASHServer/Web Content/* - This folder contains web files, such as *HTML*, *JavaScript*, *CSS*, *JSP*, etc. The user can create these files and even sub directories in this folder. In this project, there was no need for web file creation, so the directory has the default files and folders.

13. R-Click on ***Web Content*** -> ***New***->***File***, choose *Web Content* location and a file name and click ***Finish***
14. R-Click on ***Web Content*** -> ***New***->*Folder*, choose *Web Content* location and a folder name and click ***Finish***

- *DASHServer/Web Content/WEB-INF/* - This folder contains a *XML* file (*web.xml*) that is optional with *Servlets 3.0*, but required in 2.5 and earlier. In this project this file will be used.
- *Servers* – This folder is created when *Tomcat* was configured earlier. There was no need to change any of the default files in it.

5.1.3. Servlet

The *Servlet* [22] job is to read data sent by the client, generate the results and send data back to the client. This *Servlet* does in fact nothing more that obtaining an *InputStream* of the desired resource/file and writing it to the *OutputStream* of the *HTTP* response along with a set of important response headers.

1. First the *Servlet* needs to be initialized, validating the base path to get all the resources from. In this project the resources were stored in

```
public void init() throws ServletException {

    // Get base path (path to get all resources from) as init parameter.
    this.basePath = "/Users/PC/workspace/SERVER/repositorio/";

    // Validate base path.
    if (this.basePath == null) {
        throw new ServletException("FileServlet init param 'basePath' is required.");
    } else {
        File path = new File(this.basePath);
        if (!path.exists()) {
            throw new ServletException("FileServlet init param 'basePath' value '"
                + this.basePath + "' does actually not exist in file system.");
        } else if (!path.isDirectory()) {
            throw new ServletException("FileServlet init param 'basePath' value '"
                + this.basePath + "' is actually not a directory in file system.");
        } else if (!path.canRead()) {
            throw new ServletException("FileServlet init param 'basePath' value '"
                + this.basePath + "' is actually not readable in file system.");
        }
```

‘/Users/PC/workspace/SERVER/repositorio/’.

2. Then the head request will be processed in to separated ways: only the header (without the content) and with content.

```
protected void doHead(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        // Process request without content.
        processRequest(request, response, false);
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        // Process request with content.
        processRequest(request, response, true);

    }
```

3. The request file is validated, checking his existence, location and *URL*. For files catching, resume and range the headers should be processed as well.

```
protected void doHead(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        // Process request without content.
        processRequest(request, response, false);
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        // Process request with content.
        processRequest(request, response, true);

    }
    private void processRequest
        (HttpServletRequest request, HttpServletResponse response, boolean content)
            throws IOException
    {
        // Validate the requested file ---------------------------------------------

        // Get requested file by path info.
        String requestedFile = request.getPathInfo();

        // Check if file is actually supplied to the request URL.
        if (requestedFile == null) {
            // Do your thing if the file is not supplied to the request URL.
            // Throw an exception, or send 404, or show default/warning page, or just ignore it.
            response.sendError(HttpServletResponse.SC_NOT_FOUND);
            return;
        }

        // URL-decode the file name (might contain spaces and on) and prepare file object.
        File file = new File(basePath, URLDecoder.decode(requestedFile, "UTF-8"));

        // Check if file actually exists in filesystem.
        if (!file.exists()) {
            // Do your thing if the file appears to be non-existing.
            // Throw an exception, or send 404, or show default/warning page, or just ignore it.
            response.sendError(HttpServletResponse.SC_NOT_FOUND);
            return;
        }

        // Prepare some variables. The ETag is an unique identifier of the file.
        String fileName = file.getName();
        long length = file.length();
        long lastModified = file.lastModified();

        String eTag = fileName + "_" + length + "_" + lastModified;
```

```
// Validate request headers for caching ---------------------------------------------------

        // If-None-Match header should contain "*" or ETag. If so, then return 304.
        String ifNoneMatch = request.getHeader("If-None-Match");
        if (ifNoneMatch != null && matches(ifNoneMatch, eTag)) {
            response.setHeader("ETag", eTag); // Required in 304.
            response.sendError(HttpServletResponse.SC_NOT_MODIFIED);
            return;
        }

        // If-Modified-Since header should be greater than LastModified. If so, then return 304.
        // This header is ignored if any If-None-Match header is specified.
        long ifModifiedSince = request.getDateHeader("If-Modified-Since");
        if (ifNoneMatch == null && ifModifiedSince != -1 && ifModifiedSince + 1000 > lastModified) {
            response.setHeader("ETag", eTag); // Required in 304.
            response.sendError(HttpServletResponse.SC_NOT_MODIFIED);
            return;
        }


        // Validate request headers for resume ----------------------------------------------------

        // If-Match header should contain "*" or ETag. If not, then return 412.
        String ifMatch = request.getHeader("If-Match");
        if (ifMatch != null && !matches(ifMatch, eTag)) {
            response.sendError(HttpServletResponse.SC_PRECONDITION_FAILED);
            return;
        }

        // If-Unmodified-Since header should be greater than LastModified. If not, then return 412.
        long ifUnmodifiedSince = request.getDateHeader("If-Unmodified-Since");
        if (ifUnmodifiedSince != -1 && ifUnmodifiedSince + 1000 <= lastModified) {
            response.sendError(HttpServletResponse.SC_PRECONDITION_FAILED);
            return;
         }
```

```java
        // Validate and process range -------------------------------------------------------------

        // Prepare some variables. The full Range represents the complete file.
        Range full = new Range(0, length - 1, length);
        List<Range> ranges = new ArrayList<Range>();

        // Validate and process Range and If-Range headers.
        String range = request.getHeader("Range");
        if (range != null) {

            // Range header should match format "bytes=n-n,n-n,n-n...". If not, then return 416.
            if (!range.matches("^bytes=\\d*-\\d*(,\\d*-\\d*)*$")) {
                response.setHeader("Content-Range", "bytes */" + length); // Required in 416.
                response.sendError(HttpServletResponse.SC_REQUESTED_RANGE_NOT_SATISFIABLE);
                return;
            }

            // If-Range header should either match ETag or be greater then LastModified. If not,
            // then return full file.
            String ifRange = request.getHeader("If-Range");
            if (ifRange != null && !ifRange.equals(eTag)) {
                try {
                    long ifRangeTime = request.getDateHeader("If-Range"); // Throws IAE if invalid.
                    if (ifRangeTime != -1 && ifRangeTime + 1000 < lastModified) {
                        ranges.add(full);
                    }
                } catch (IllegalArgumentException ignore) {
                    ranges.add(full);
                }
            }

            // If any valid If-Range header, then process each part of byte range.
            if (ranges.isEmpty()) {
                for (String part : range.substring(6).split(",")) {
                    // Assuming a file with length of 100, the following examples returns bytes at:
                    // 50-80 (50 to 80), 40- (40 to length=100), -20 (length-20=80 to length=100).
                    long start = sublong(part, 0, part.indexOf("-"));
                    long end = sublong(part, part.indexOf("-") + 1, part.length());

                    if (start == -1) {
                        start = length - end;
                        end = length - 1;
                    } else if (end == -1 || end > length - 1) {
                        end = length - 1;
                    }

                    // Check if Range is syntactically valid. If not, then return 416.
                    if (start > end) {
                        response.setHeader("Content-Range", "bytes */" + length); // Required in 416.
                        response.sendError(HttpServletResponse.SC_REQUESTED_RANGE_NOT_SATISFIABLE);
                        return;
                    }

                    // Add range.
                    ranges.add(new Range(start, end, length));
                }

            }
```

4. Now a response is prepared and initialized. This response consists on sending the file parts to the client.

```
// Send requested file (part(s)) to client ------------------------------------------------

        // Prepare streams.
        RandomAccessFile input = null;
        OutputStream output = null;

        try {
            // Open streams.
            input = new RandomAccessFile(file, "r");
            output = response.getOutputStream();

            if (ranges.isEmpty() || ranges.get(0) == full) {

                // Return full file.
                Range r = full;
                response.setContentType(contentType);
                response.setHeader("Content-Range", "bytes " + r.start + "-" + r.end + "/" + r.total);

                if (content) {
                    if (acceptsGzip) {
                        // The browser accepts GZIP, so GZIP the content.
                        response.setHeader("Content-Encoding", "gzip");
                        output = new GZIPOutputStream(output, DEFAULT_BUFFER_SIZE);
                    } else {
                        // Content length is not directly predictable in case of GZIP.
                        // So only add it if there is no means of GZIP, else browser will hang.
                        response.setHeader("Content-Length", String.valueOf(r.length));
                    }

                    // Copy full range.
                    copy(input, output, r.start, r.length);
                }

            } else if (ranges.size() == 1) {

                // Return single part of file.
                Range r = ranges.get(0);
                response.setContentType(contentType);
                response.setHeader("Content-Range", "bytes " + r.start + "-" + r.end + "/" + r.total);
                response.setHeader("Content-Length", String.valueOf(r.length));
                response.setStatus(HttpServletResponse.SC_PARTIAL_CONTENT); // 206.

                if (content) {
                    // Copy single part range.
                    copy(input, output, r.start, r.length);
                }
```

5.2. Client Side

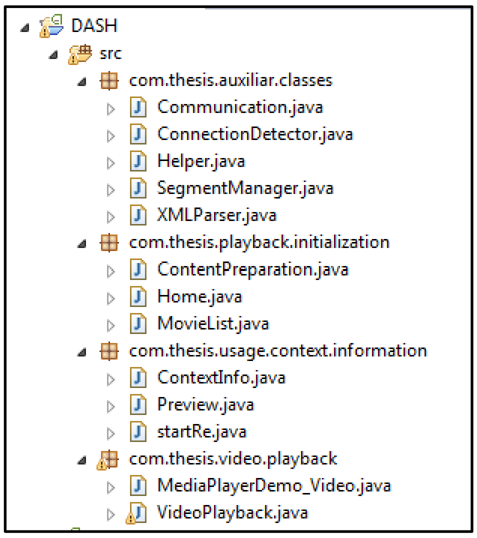
In the client-server relationship, the Client is the one who performs the requests and receives data from those requests, send by the Server. This specific Client is an *Android* application, which requires Internet connection to be able to perform *HTTP Adaptive Streaming*, using *MPEG-DASH* as protocol.

MPEG-DASH doesn’t specify anything about the Client side. As long as it can interpret the protocol correctly, the *DASH* Client could be developed for any platform. And because it is not fully released, other tools beside the recommended ones could be used to generate the content. These other tools could be used as long as it can generate the content according to *DASH* specifications.

It was developed an *Android* application that simulates a simple *VoD* client that uses *MPEG-DASH* as protocol for the video streaming. The Client has access to a list of movies available in the Server for streaming. He can choose a video and it will start playing as soon as possible.

5.2.1. Project Structure

The project developed consists of three different stages: *Playback Initialization*, *Video Playback* and *Usage Context Information*. Each stage is represented by packages. Each package contains the classes that play a direct role in his functionality. Additionally, there is one more package, *Auxiliar Classes*, which contains the classes that play an indirect role in all other classes mentioned before.



```
DASH
  src
    com.thesis.auxiliar.classes
      Communication.java
      ConnectionDetector.java
      Helper.java
      SegmentManager.java
      XMLParser.java
    com.thesis.playback.initialization
      ContentPreparation.java
      Home.java
      MovieList.java
    com.thesis.usage.context.information
      ContextInfo.java
      Preview.java
      startRe.java
    com.thesis.video.playback
      MediaPlayerDemo_Video.java
      VideoPlayback.java
```

Figure 35 - *DASH Client* structure.

The first stage is the *Playback Initialization*, on which the application establishes a connection to the Server, download and parse the metadata and prepare the initial media content to be played.

The second stage is the *Video Playback*, where the application plays downloaded content and download new one to be played.

The third stage is *Usage Context Information*, on which context information is collected while the media content is being played.

5.2.2. Playback Initialization

The *Playback Initialization* is very important, mainly because of the metadata parse and the media content preparation. Without the metadata it would be impossible for the application to know here the video segments are located. This package is constituted by three major activities: *Home*, *MovieList* and *ContentPreperation*.

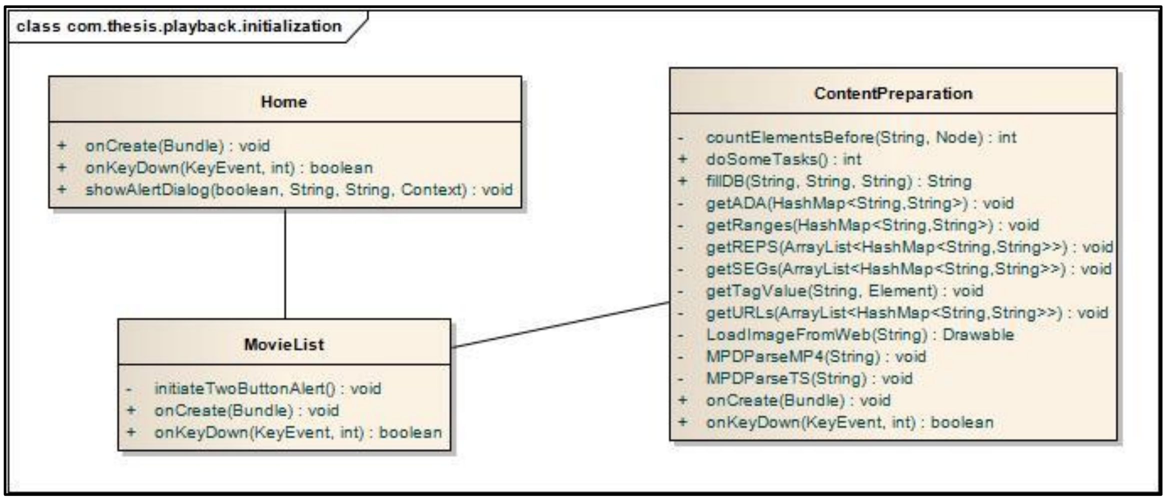

class com.thesis.playback.initialization
Home
+ onCreate(Bundle) : void
+ onKeyDown(KeyEvent, int) : boolean
+ showAlertDialog(boolean, String, String, Context) : void
MovieList
- initiateTwoButtonAlert() : void
+ onCreate(Bundle) : void
+ onKeyDown(KeyEvent, int) : boolean
ContentPreparation
- countElementsBefore(String, Node) : int
+ doSomeTasks() : int
+ fillDB(String, String, String) : String
- getADA(HashMap<String,String>) : void
- getRanges(HashMap<String,String>) : void
- getREPS(ArrayList<HashMap<String,String>>) : void
- getSEGs(ArrayList<HashMap<String,String>>) : void
- getTagValue(String, Element) : void
- getURLs(ArrayList<HashMap<String,String>>) : void
- LoadImageFromWeb(String) : Drawable
- MPDParseMP4(String) : void
- MPDParseTS(String) : void
+ onCreate(Bundle) : void
+ onKeyDown(KeyEvent, int) : boolean

Figure 36 - *UML* class diagram of *Playback Initialization*.

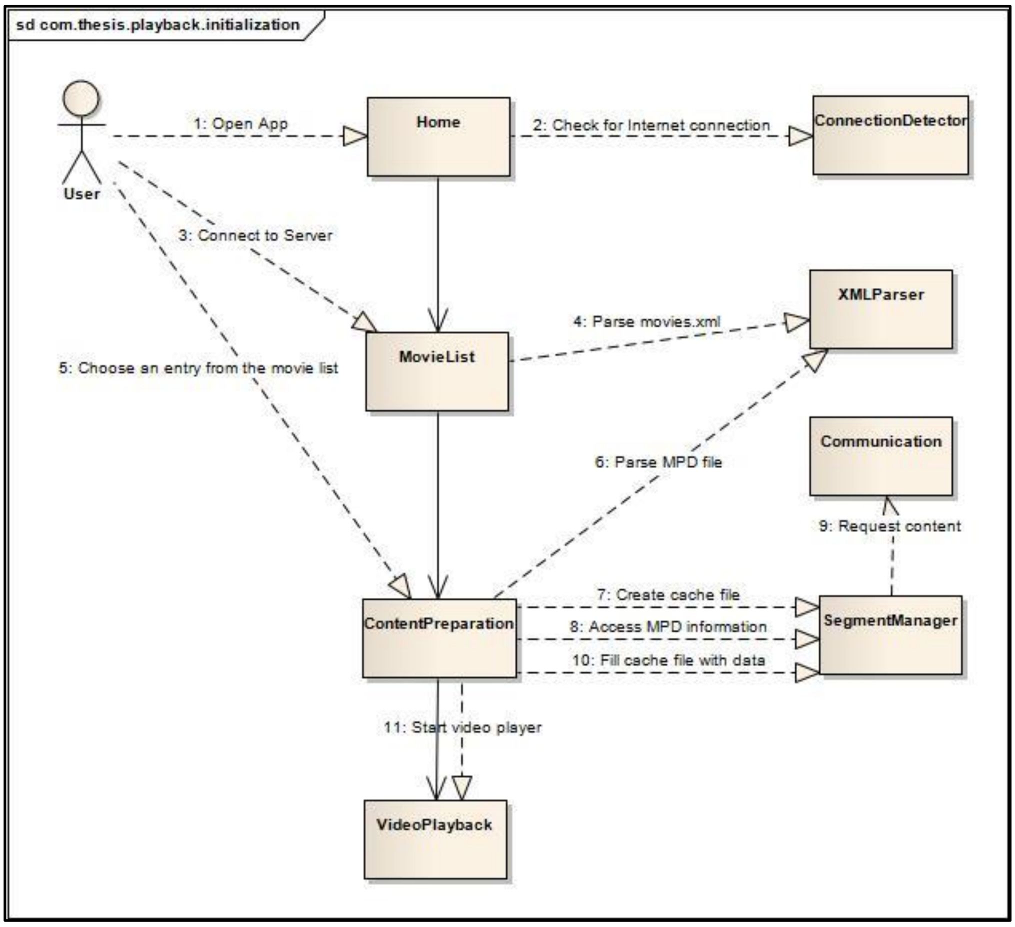


Figure 37 - *UML* collaboration diagram of *Playback Initialization*.

1. Home

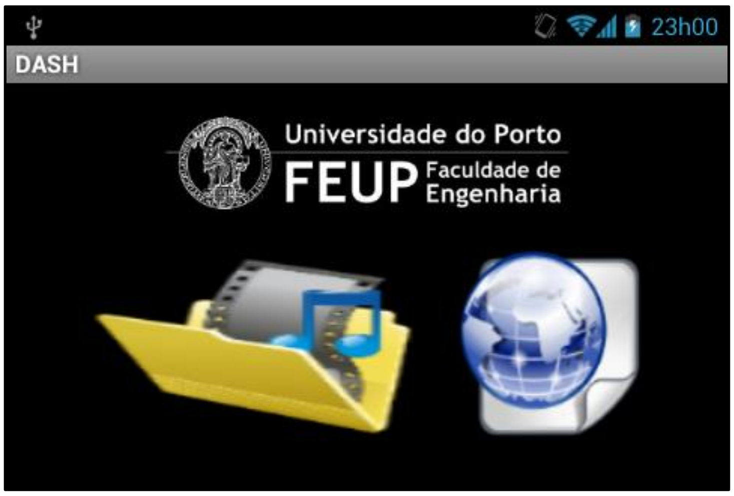


Figure 38 - *Home* layout.

This activity is the home screen of the application. The client is presented with the choice of continue with the application or visit the website of the project. The last choice will open the default browser of the *Android* device. If the client chooses to continue the application, it will be redirect to the next activity, *MovieList*.

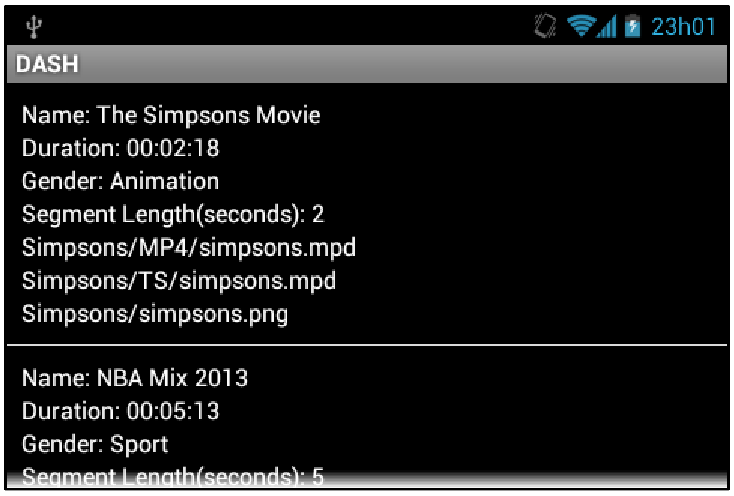
The activity starts to check is there is Internet connection on the device. If so, it will connect to the server, by requesting the metadata file, *movies.xml*, which is used to build the *MovieList* activity.

To check if the device has Internet connection and to establish connection to the Server, the code bellow can be used. If there isn't a valid Internet connection, a dialog box will appear to inform the client. The application won't let the user do anything until he has a valid connection. The same will append if he has Internet connection, but he can't reach *movies.xml*, which most likely means that the Server is down.

```
// Check connection to Internet -----------------------------------------

 public boolean isConnectingToInternet(){
        ConnectivityManager connectivity = (ConnectivityManager)
_context.getSystemService(Context.CONNECTIVITY_SERVICE);
          if (connectivity != null)
          {
              NetworkInfo[] info = connectivity.getAllNetworkInfo();
              if (info != null)
                  for (int i = 0; i < info.length; i++)
                      if (info[i].getState() == NetworkInfo.State.CONNECTED)
                      {
                          return true;
                      }
          }
          return false;

    }
```

```
// Check connection to Server ----------------------------------------

URL url = new URL( "http://"+IP+"/DASHServer/file/movies.xml" );
                HttpURLConnection httpConn =  (HttpURLConnection)url.openConnection();
                httpConn.setInstanceFollowRedirects( false );
                httpConn.setRequestMethod( "HEAD" );

                httpConn.connect();
```

2. MovieList

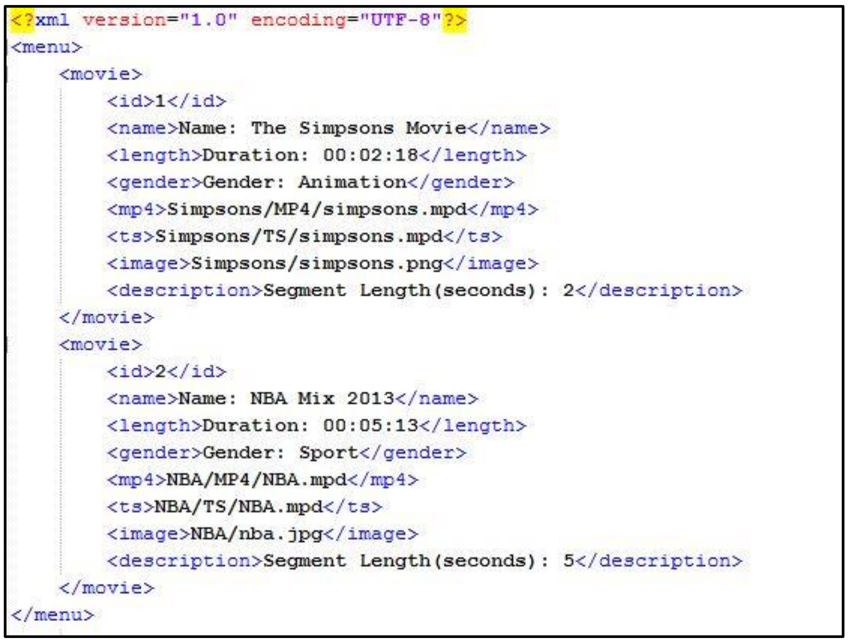


Name: The Simpsons Movie
Duration: 00:02:18
Gender: Animation
Segment Length(seconds): 2
Simpsons/MP4/simpsons.mpd
Simpsons/TS/simpsons.mpd
Simpsons/simpsons.png

Name: NBA Mix 2013
Duration: 00:05:13
Gender: Sport
Segment Length(seconds): 5

Figure 39 - *MovieList* layout.

This activity uses a *ListView* to present a list of available movies on the server to be streamed. The information presented results from parsing *movies.xml*. Information such as movie name, duration, gender, segment length, movie metadata and poster location are presented to the client.



```
<?xml version="1.0" encoding="UTF-8"?>
<menu>
    <movie>
        <id>1</id>
        <name>Name: The Simpsons Movie</name>
        <length>Duration: 00:02:18</length>
        <gender>Gender: Animation</gender>
        <mp4>Simpsons/MP4/simpsons.mpd</mp4>
        <ts>Simpsons/TS/simpsons.mpd</ts>
        <image>Simpsons/simpsons.png</image>
        <description>Segment Length(seconds): 2</description>
    </movie>
    <movie>
        <id>2</id>
        <name>Name: NBA Mix 2013</name>
        <length>Duration: 00:05:13</length>
        <gender>Gender: Sport</gender>
        <mp4>NBA/MP4/NBA.mpd</mp4>
        <ts>NBA/TS/NBA.mpd</ts>
        <image>NBA/nba.jpg</image>
        <description>Segment Length(seconds): 5</description>
    </movie>
</menu>
```

Figure 40 - *movies.xml* file structure.

The activity downloads the movies.xml file and parses it to display his content on the screen. When the client selects an entry from the list, it will be given a choice to request the total length of the content or use partial requests. Either case, the client is redirected to the next activity, *ContentPreparation*.

```
// Download movies.xml -----------------------------------------
    public String getXmlFromUrl(String url) {
       String xml = null;
       Document doc = null;
        DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();

        try {
            // defaultHttpClient
            DefaultHttpClient httpClient = new DefaultHttpClient();
           // HttpPost httpPost = new HttpPost(url);
            HttpGet httpGet = new HttpGet(url);

            HttpResponse httpResponse = httpClient.execute(httpGet);
            HttpEntity httpEntity = httpResponse.getEntity();
            xml = EntityUtils.toString(httpEntity);


            DocumentBuilder db = dbf.newDocumentBuilder();
            InputSource is = new InputSource();
                is.setCharacterStream(new StringReader(xml));
                doc = db.parse(is);

        } catch (UnsupportedEncodingException e) {
            e.printStackTrace();
        } catch (ClientProtocolException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

        return doc;

    }
```

To download the *movies.xml* the application makes a request to the Server and receives a response. The following code can be used for a simple *XML* file request-response process.

To parse it the application gets the value of an element with a specific name and save both on a *HASH-MAP*.

```
// Save data from movies.xml -----------------------------------------
NodeList nl = doc.getElementsByTagName(KEY_MOVIE);
        // looping through all item nodes <item>
        for (int i = 0; i < nl.getLength(); i++) {
            // creating new HashMap
            HashMap<String, String> map = new HashMap<String, String>();
            Element e = (Element) nl.item(i);
            // adding each child node to HashMap key => value
            map.put(KEY_ID, parser.getValue(e, KEY_ID));
            map.put(KEY_NAME, parser.getValue(e, KEY_NAME));
            map.put(KEY_LEN, parser.getValue(e, KEY_LEN));
            map.put(KEY_GEN, parser.getValue(e, KEY_GEN));
            map.put(KEY_MP4, parser.getValue(e, KEY_MP4));
            map.put(KEY_TS, parser.getValue(e, KEY_TS));
            map.put(KEY_IMA, parser.getValue(e, KEY_IMA));
            map.put(KEY_DESC, parser.getValue(e, KEY_DESC));

            // adding HashList to ArrayList
            menuItems.add(map);

        }
```

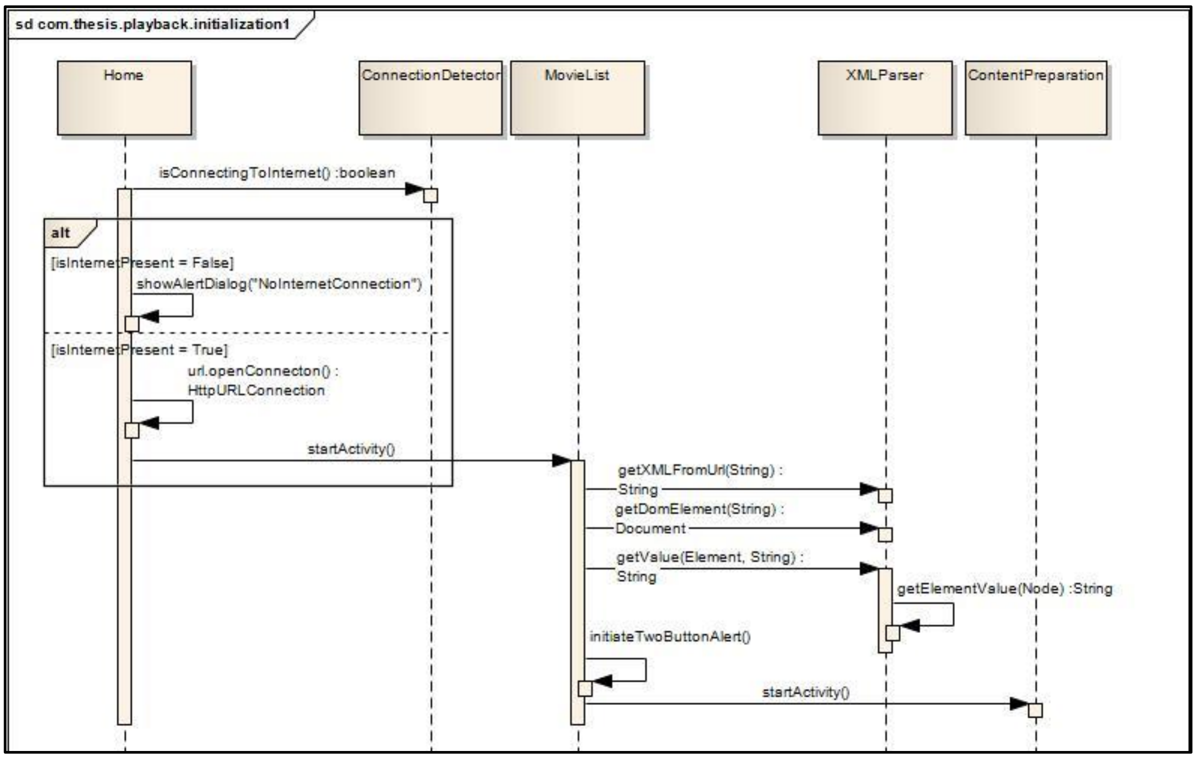
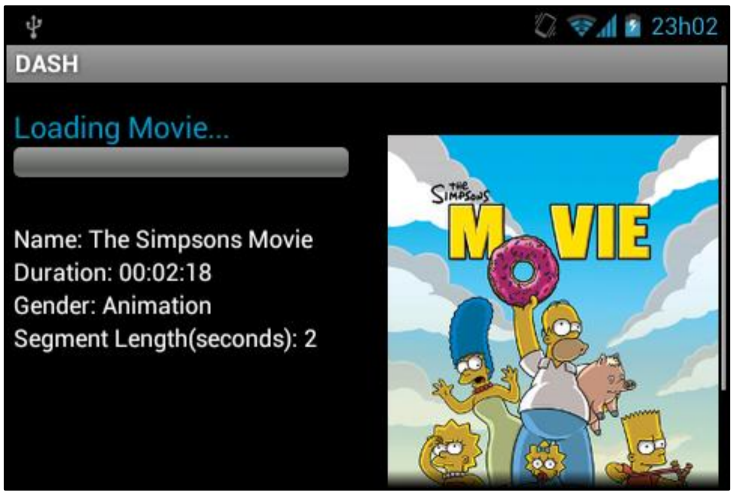

sd com.thesis.playback.initialization1
Home
ConnectionDetector
MovieList
XMLParser
ContentPreparation
isConnectingToInternet() :boolean
alt
[isInternetPresent = False]
showAlertDialog("NoInternetConnection")
[isInternetPresent = True]
url.openConnecton() : HttpURLConnection
startActivity()
getXMLFromUrl(String) : String
getDomElement(String) : Document
getValue(Element, String) : String
getElementValue(Node) :String
initiateTwoButtonAlert()
startActivity()

Figure 41 - *UML* sequence diagram of *Home* and *MovieList*.

3. ContentPreparation



Loading Movie...

Name: The Simpsons Movie
Duration: 00:02:18
Gender: Animation
Segment Length(seconds): 2

Figure 42 - *ContentPreparation* layout.

This activity results from the selection of a movie from the list. It displays movie information, such as name, duration, gender, segment length and the movie poster. This information is displayed while the application prepares the movie for playback. A progress bar is also shown to represent the time it takes for this preparation. The application must:

- Download the correct *MPD* file of the movie selected.
- Parse the *MPD* information.
- Download a few seconds of movie data.
- Save the data into a cache.

When the cache file is ready, the client is redirect to *Video Playback* activity and start the playback. The initial data downloaded acts as a buffer. His length is big enough to be able to start the media content playback and small enough to start the playback as fast as it can. The buffer's length in this case is 5 seconds.

```
// Parse and save MPD content related to Adaptation Set element -------------------------
        HashMap<String, String> ADA = new HashMap<String, String>();
         NodeList n2 = doc.getElementsByTagName("Period");
            NodeList n3 = doc.getElementsByTagName("AdaptationSet");
            NodeList n5 = doc.getElementsByTagName("Representation");
            NodeList n7 = doc.getElementsByTagName("RepresentationIndex");
            NodeList n9 = doc.getElementsByTagName("SegmentList");
            NodeList n10 = doc.getElementsByTagName("SegmentURL");

                Element e2 = (Element) n2.item(0);
                 getTagValue(e2,"dB");
               Element e3 = (Element) n3.item(0);
                 getTagValue(e3,"ts");

                 ADA.put("duration",PERduration);
                 ADA.put("bitstreamSwitching",ADAbitstreamSwitching);
                 ADA.put("segmentAlignment",ADAsegmentAlignment);
                 ADA.put("subsegmentAlignment",ADAsubsegmentAlignment);
                   ADA.put("maxWidth",ADAmaxWidth);     ADA.put("maxHeight",ADAmaxHeight);
```

The methods used for download and parse *movies.xml* are used on the *MPD* files as well. The parsing is a little trickier, because it doesn't have a linear structure as *movie.xml* does. This parsing results on *HASH-MAPs* for *Adaptation Sets*, *Representations* and *URLs* of the media segments. The tag name of the *HASH-MAP* is the respective element's attribute and the value is the attribute' value.

```
// Store segments into a temporary file -------------------------
public static File FillCache(ArrayList<String> strArray2, File c, String name) throws IOException{
        File bufferedFile = null;

         bufferedFile = File.createTempFile(name, ".dat");
         File tmpFile = null;
         String[] stockArr = new String[strArray2.size()];
            stockArr = strArray2.toArray(stockArr);

                for (int i=0; i<strArray2.size(); i++) {

                tmpFile = com.getDataSource(stockArr[i],c);
                Log.v(TAG, "getdatasource:"+i);

                bufferedFile = moveFile(tmpFile, name);
                }
                float bytesRead = tmpFile.length();

                Log.v(TAG, "Writting on cache complete");
                return bufferedFile;

        }
```

First a temporary *.dat* file needs to be created. Then, the application has to request the segments and download them into a cache file, using the *getDataSource* method. Finally, the *moveFile* method creates an exact copy of the temporary file and appends it to *bufferedFile*. This file is the actual file being played and it will be appended more data to it every time *FillCache* is called. To use partial requests, *retreiveSegmentContent* is called to an *InputStream*.

```
// Get data from segment files -------------------------
public File getDataSource(String path, File temp) throws IOException {
                        System.out.println("Path:"+path);

                        URL url = new URL(path);
                        URLConnection cn = url.openConnection();
                        cn.connect();
                        InputStream stream = cn.getInputStream();

                        if (stream == null)
                                throw new RuntimeException("stream is null");

                String tempPath = temp.getAbsolutePath();
                FileOutputStream out = new FileOutputStream(tempPath);


                byte buf[] = new byte[1024];
                do {
                        int numread = stream.read(buf);
                        if (numread < 0){
                                break;}
                        out.write(buf, 0, numread);
                } while (true);
                try {
                        stream.close();
                        out.close();
                } catch (IOException ex) {
                        Log.e(TAG, "error: " + ex.getMessage(), ex);
                }

                return temp;
```

```
// Request segment using byte range -------------------------
private InputStream retreiveSegmentContent(String specificURL, int startByte, int endByte) {
                HttpClient httpClient = new DefaultHttpClient();
                HttpGet getSegment = new HttpGet(specificURL);
                System.out.println("bytes=" + startByte + "-" + endByte);
                getSegment.addHeader("Range", "bytes=" + startByte + "-" + endByte);
                InputStream content = null;
                        HttpResponse response = httpClient.execute(getSegment);
                        HttpEntity entity = response.getEntity();
                        Log.d(TAG, "Status code: " + response.getStatusLine().getStatusCode());
                        content = entity.getContent();
                return content;
        }
```

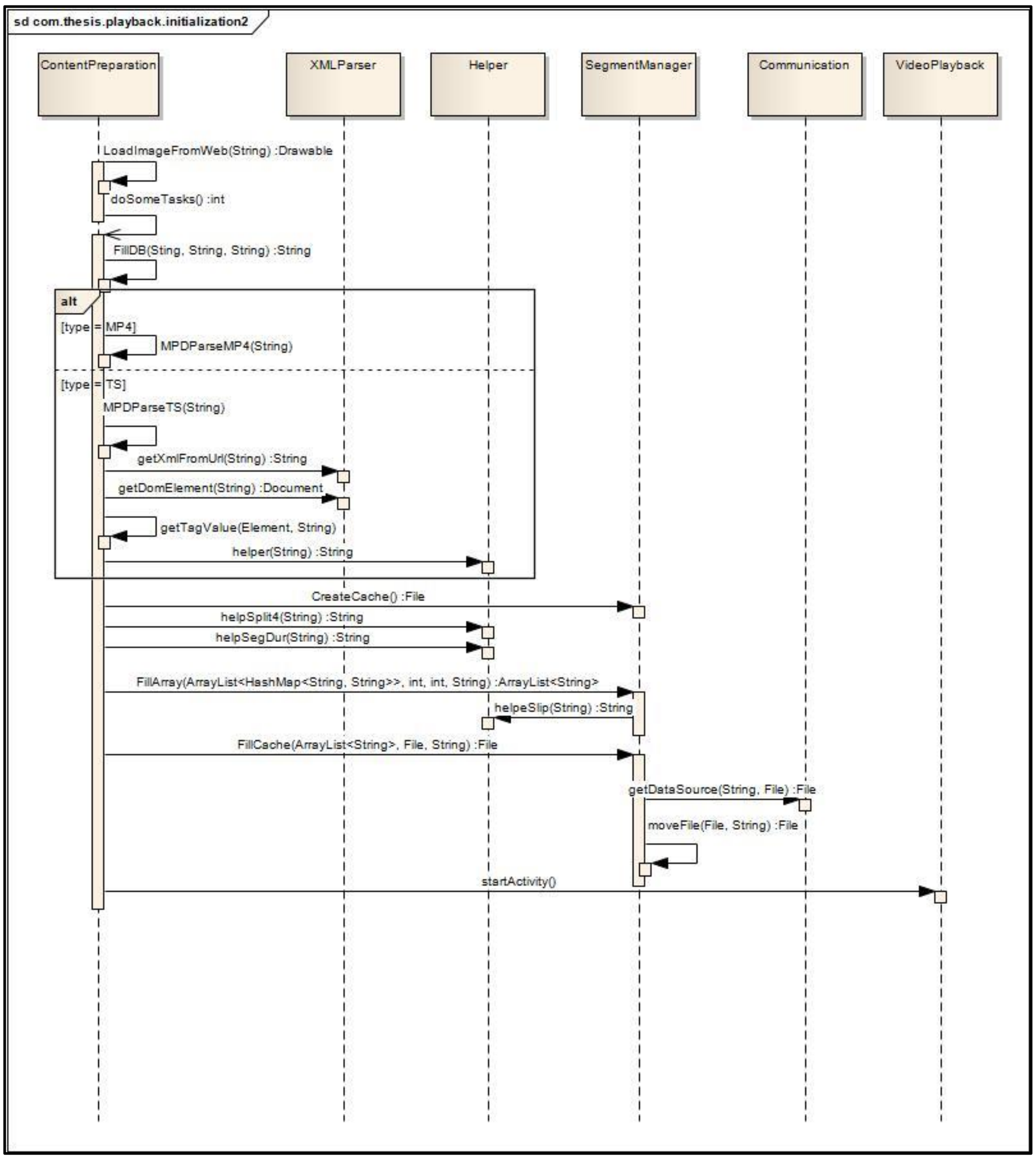

sd com.thesis.playback.initialization2
ContentPreparation
XMLParser
Helper
SegmentManager
Communication
VideoPlayback
LoadImageFromWeb(String) :Drawable
doSomeTasks() :int
FillDB(Sting, String, String) :String
alt
[type = MP4]
MPDParseMP4(String)
[type = TS]
MPDParseTS(String)
getXmlFromUrl(String) :String
getDomElement(String) :Document
getTagValue(Element, String)
helper(String) :String
CreateCache() :File
helpSplit4(String) :String
helpSegDur(String) :String
FillArray(ArrayList<HashMap<String, String>>, int, int, String) :ArrayList<String>
helpeSlip(String) :String
FillCache(ArrayList<String>, File, String) :File
getDataSource(String, File) :File
moveFile(File, String) :File
startActivity()

Figure 43 - *UML* sequence diagram of *ContentPreparation*.

5.2.3. Video Playback

Video Playback is where the user can actually watch the movie. This stage is constituted by only one activity, which specifies a *VideoView* to display the downloaded media content. Besides that, is in this activity that the methods for usage context information are called. They actually run in parallel threads alongside with the video playback.

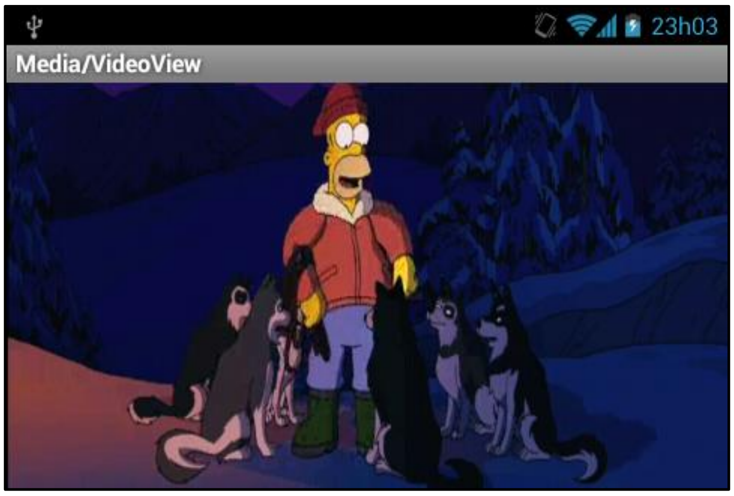


Figure 44 - *VideoPlayback* layout.

The activity is called the moment the content preparation is done. After the cache file is open for playback, the application must continue downloading the next movie segments. It uses the same process as before: download, save on a temporary file and copy the content into the file that is being played. If the cache doesn't get filled as fast at download new content, the reading of the file content will come to an end and the playback will stop.

```
// Control segment count -------------------------
private void begin(final boolean first) throws IOException, InterruptedException {
                 int seg_ini =  Integer.parseInt(seg_dur);
                 System.out.println("seg_ini: "+seg_ini);
                 if(first){
                         Next_Seg = seg_ini;
                 }else{
                         Next_Seg ++;
                 }
                 System.out.println("Next_Seg: "+Next_Seg);
        try {
                        downloadingMediaFile = SegmentManager.CreateCache();
                        startStreaming(urls,SegmentManager.ReturnID(),1,Next_Seg);
                } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();

                }
```

Control which segment has to be downloaded next, to maintain the order of movie segments, can be done with the counter *Next_Seg*. In the beginning, *Next_Seg* equals the last segment stores in the cache on the preparation stage. From that point forward, it will increment every time that the next segment needs to be downloaded, until it reaches the total count of movie segments.

```
    // Get the data content all store in cache -------------------------
public void startStreaming(final ArrayList<HashMap<String, String>> urls2, final String rep, final int
num_Seg, final int count_Seg) throws IOException {

        if(!cache_ready){

        mediaUrl = SegmentManager.FillArray(urls,num_Seg,count_Seg, FillURL);
        cache_ready = true;
Log.v(TAG, "Cache ready "+cache_ready);

        }else{

        SegmentManager.FillCache(mediaUrl,downloadingMediaFile,cacheName);
        Log.v(TAG, "FillCache Complete ");
        cache_ready = false;
Log.v(TAG, "Cache ready "+cache_ready);

}
```

5.2.4. Usage Context Information

Usage Context Information, as the name indicates, is the stage where context information is collected. Based on this information, the application takes some action to adapt the video playback. The context information is collected from many sources, relying on *Android* sensors and resources to do the job.

There are many strategies to collect this kind of information, there isn't the right or wrong way, but there is a better and a worst way. Obviously, if the measures need to be precise, more complex methods will be used and the device performance will decrease. The strategies used were the ones which returned good results for want the application needed to do, without compromising to much the device's performance.

5.2.4.1. Network Conditions

Network conditions are actually the major factor to adapt the video playback. It's awful when the user is watching a streaming session and the playback stops to rebuffer every time the available bandwidth of the network is too low to download the video file.

Every time that a segment needs to be downloaded, the application calculates the current available bandwidth. The value calculated is used as the reference of the maximum bandwidth that the segment to be downloaded can have.

The application compares every bandwidth of every representation with the maximum bandwidth and selects the closest value, without surpass the calculated value. The selected representation is then used in the request segment process.

```
    // Get all representations IDs and respective bandwidth -------------------------
public static void Retrive(final ArrayList<HashMap<String, String>> Representacoes, final String
NumReps) {

        String bandwidth = "";
        int numreps=Integer.parseInt(NumReps);//System.out.println("numreps"+numreps);
        String[] reps = new String[numreps];
        int[] Reps = new int[numreps];
        long[] Bands = new long[numreps];

        for(int i=0;i<numreps;i++){
                String aux = Integer.toString(i+1);
                reps[i] = aux;
                Reps[i] = i+1;
                    for (HashMap<String, String> map : Representacoes){

                        if(map.get("id").equals(reps[i])){
                        bandwidth = map.get("bandwidth");
                        Bands[i] = Long.parseLong(bandwidth);
                        }}
```

```
    // Compare bandwidths and select the correct representation ID ----------------
private static void compareBand(int[] reps, long[] bands) {
                        long nearest = -1;
                        float bestDistanceFoundYet = Integer.MAX_VALUE;

                        for (int i = 0; i < bands.length; i++) {
                          if (bands[i] == Float.parseFloat(Bandwidth)) {
                                  ID = Integer.toString(reps[i]);
                          } else {
                            float d = Float.parseFloat(Bandwidth) - bands[i];
                            if (d < bestDistanceFoundYet && d>0) {
                              nearest = bands[i];
                              bestDistanceFoundYet = d;
                              ID = Integer.toString(reps[i]);
                            }}}
```

5.2.4.2. Noise Conditions

A noisy environment, when the use is trying to watch a streaming session without headphones, can ruin the whole experience. The user may try guiding himself from subtitles, but his perception of the movie is not as great as it should be.

At the beginning of playback, a new thread *startRe* is called. This class accesses the device's microphone and starts a new audio recording. In this recording, the maximum amplitude of the file is retrieved 10 times separated by 200ms of each other, and then stop recording. This process is repeated 10 times, separated in equal intervals, until the end of the playback.

```
    // Calculate noise level from audio recording ----------------
public double getNoiseLevel() {
        Log.d("SPLService", "getNoiseLevel() ");
   double x = mRecorder.getMaxAmplitude();
    x2 = EMA_FILTER * x +(1.0 - EMA_FILTER) * x2;
    double pressure = x2/51805.5336;

  //  Log.d("SPLService", "x="+x);
    double db = (20 * Math.log10(pressure / REFERENCE));
    Log.d("SPLService", "db="+db);
    if(db>0)
    {
        return db;
    }
    else
    {
        return 0;
    }
}
```

To calculate the noise level from the audio recording, the application gets the maximum amplitude of the audio recording and returns the value in *dB*.

```
    // Change volume of playback ----------------
public static void changeVolume(int lev, AudioManager audio) {

        if(lev==1){

                if(!Vol_Up && !Vol_Down){
                audio.adjustStreamVolume(AudioManager.STREAM_MUSIC,
                    AudioManager.ADJUST_RAISE, AudioManager.FLAG_PLAY_SOUND);
            Log.i("Audio Raise", "");
        }
        }

        if(lev==2){

                if(!Vol_Up && !Vol_Down){
                audio.adjustStreamVolume(AudioManager.STREAM_MUSIC,
                    AudioManager.ADJUST_LOWER, AudioManager.FLAG_PLAY_SOUND);
                Log.i("Audio Lower", "");
        }

        }
```

To change the volume of playback, the application needs to compare the current measure to the one made before. If the noise level increased, *'int lev = 1'* and the application will increase the audio playback, else *'int lev = 2'* and the application will decrease the audio playback. This is done if the buttons for volume up and volume down of the device weren't used, because if they were in fact used, means that the user wants to adjust audio volume manually.

5.2.4.3. Brightness Conditions

The device's screen brightness is important to the user experience and should be adapted according to environment light conditions. On a sunny day, is the user is outside with his phone, he has to increase the screen brightness otherwise he will have trouble seeing what is presented. On the other hand, if he is in a very dark room, there is no need to have a very bright screen. He may cause discomfort to his eyes and the battery's life time will decrease.

So, screen brightness adaptation happens to be a big deal when it comes to give the best user experience possible during video playback.

Some *Android* devices incorporate a light sensor that detects the level of brightness in the surrounding room. When it comes to screen brightness adaptation, this sensor should be used. If the device doesn't have one, an alternative method can be used, which call on the device's camera. If the device has no camera, then brightness adaptation won't be able using this application.

```
// Check if there is a light sensor and a camera ----------------
Sensor lightSensor = sensorManager.getDefaultSensor(Sensor.TYPE_LIGHT);
if (lightSensor == null){

if (!getPackageManager()        .hasSystemFeature(PackageManager.FEATURE_CAMERA)) {

System.out.println("No Camera in this device");
        } else{
preview = new Preview(this);
 ((FrameLayout) findViewById(R.id.preview)).addView(preview);
System.out.println("Camera preview ON");
}
}else{
max =  lightSensor.getMaximumRange();
sensorManager.registerListener(lightSensorEventListener,  lightSensor,
SensorManager.SENSOR_DELAY_NORMAL);

}
```

```
        // Light sensor event listener----------------
 public void onSensorChanged(SensorEvent event) {
                  if(event.sensor.getType()==Sensor.TYPE_LIGHT){
                   currentReading = event.values[0];

                   float light = currentReading/max;//0-1

                WindowManager.LayoutParams params =

                            getWindow().getAttributes();
                            params.flags |= LayoutParams.FLAG_KEEP_SCREEN_ON;
                            params.screenBrightness = light; //0-1
                            getWindow().setAttributes(params);

                  }}
```

If a light sensor exists on the device, an *Event Listener* will be trigger every time the light conditions change and get the application measures the current brightness value. Next, the application converts that value to a value between 0 and 1 (by dividing by the maximum value that can be measure) and set the resulting value as the screen value to be.

If a light sensor doesn't exist, the application tries to use the camera for this. Manipulating the camera *API* can be a little tricky. First of all, the camera simple doesn't work if a preview screen isn't shown to the user. So, using the camera on background wasn't something that the *Android* developers considered when they developed this *API*. That's why it was created a fake *FrameLayout* to display the preview. The fake preview was resized to 1x1, so the user can see it on the screen, even though it is there.

A picture will be taken or a video record will start triggered only by user interaction. Automatic trigger didn't work. So, to trigger the camera, the user presses the menu button of the device, as seen in the code below.

```
// Take picture if the menu button is pressed ----------------
case KeyEvent.KEYCODE_MENU:
                Log.d(TAG, "onClick ready");
                preview.camera.takePicture(null, null, jpegCallback);

                return super.onKeyDown(keyCode, event);
```

```
// Save picture taken  ----------------
public void onPictureTaken(byte[] data, Camera camera) {

        FileOutputStream outStream = null;
        String path = String.format("/sdcard/%d.jpg", System.currentTimeMillis());

        outStream = new FileOutputStream(path);
        outStream.write(data);
        outStream.close();
        Log.d(TAG, "onPictureTaken - wrote bytes: " + data.length);

        checkLight(path);

        camera.startPreview();
        Log.d(TAG, "onPictureTaken - jpeg");
}};
```

When a picture is taken, the application saves the picture in the *sdcard* and calls the *checkLight* method that will analyze the picture.

```
// Analyze picture taken  ----------------
private void checkLight(String _path) {
BitmapFactory.Options options = new BitmapFactory.Options();
Bitmap bitmap = BitmapFactory.decodeFile( _path, options );

int photoW = options.outWidth;    int photoH = options.outHeight;
int A, R = 0, G = 0, B = 0;       int r,g,b; int pixel;

                           // scan through all pixels
                           for(int x = 0; x < photoW; ++x) {
                               for(int y = 0; y < photoH; ++y) {
                                   // get pixel color
                                   pixel = bitmap.getPixel(x, y);
r = Color.red(pixel);  g = Color.green(pixel);  b = Color.blue(pixel);
R=R+r; G=G+g; B=B+b;
                         }}
R= R/(photoW * photoH);  G= G/(photoW * photoH); B= B/(photoW * photoH);
double E1 = 0.241*R*R; double E2 = 0.691*G*G;   double E3 = 0.068*B*B;
double brightness= Math.sqrt((E1)+(E2)+(E3))/255.0f;

System.out.println("brightness(0-1): "+brightness);

        WindowManager.LayoutParams params =

        getWindow().getAttributes();
        params.flags |= LayoutParams.FLAG_KEEP_SCREEN_ON;
        params.screenBrightness = (float) brightness; //0-1
        getWindow().setAttributes(params);
        Log.d(TAG, "Brightness changed");        }
```

To analyze the picture taken, the application converts it to a bitmap and gets the width and height to be able to scan through all pixels. The application saves the quantity of red, green and blue of each pixel and uses these values to calculate the brightness of the picture. The value calculated is used to set the screen brightness the same way as before with the light sensor.

5.2.4.4. Object Detection

As said before, object detection can be important in many ways: if the camera detects a world famous building, this will denunciate our location at that time. This information may be used to change the audio or the subtitles of the video to another language. As long as the picture taken is saved and the *Android API* allows this, the photo can be used to do all kinds of image processing for countless purposes.

It was used *FaceDetector.Face* class of *Android* that detects faces on a picture. This is interesting, because it can be used if there are any issues about privacy or other related with more than one person in the room.

```
// Face detection  ----------------
public void setFace(Bitmap bitmap, int photoW, int photoH) {
        FaceDetector fd;
        FaceDetector.Face [] faces = new FaceDetector.Face[MAX_FACES];
        int count = 0;

        fd = new FaceDetector(photoW, photoH, MAX_FACES);
        count = fd.findFaces(bitmap, faces);

        if(count >0){
        //Take action for privacy issues
        }
        Log.d(TAG, "Faces detected= "+count);
}
```

The application counts the number of faces detected in the photo taken. If there are one or more faces, the application can take action for privacy issues (not implemented).

Chapter 6

Tests and Conclusion

6.1. Tests

The following tests were used to evaluate the performance of the developed algorithms and the overall video playback. The performance is appraised by objective and subjective measurements.

The tests were done using a *LG Optimus One (P500)* [26], *Android* version 4.2.2 and *CPU* version *ARMv6*, 600 MHz single core and primary camera with 3.15MP, *3G* network data speed *HSDPA* 7.2 Mbps. This is the default device used to run the tests.

6.1.1. Network Conditions

The bandwidth consumption on a network can easily increase, if there are a lot of other machines using the network at same time. This means that the available bandwidth will decrease. The tests toke place on *FEUP's* network, during a work day, and in a home network with a 30Mbps connection speed. On contrary of *FEUP's* network, the home network has very little parallel bandwidth consumption, because there are no more users using the same network.

To estimate the available bandwidth it is downloaded a text file with 50 Kb located on the server. The ratio between the file size and the time it takes to download it is an approximation of the available bandwidth at that current time. The result can be more accurate if this file size increases. But, if the file size is too big, it will affect the result, because the download is consuming more bandwidth that is supposed to, decreasing the available bandwidth.

So, other method used for this measurement consists on using the last segment downloaded instead of a 50 Kb text file. The difference between the two methods is that the segments differ in size from each other. They can go from 25 Kb, such as the segments with 2 seconds duration and minimum bit rate, to 4 Mb, such as segments with 12 seconds and maximum bit rate.

Other important aspect is the fact that the Server is running in the same network as the *DASH Client*. Accessing the Server from an external network would require *VPN* connection. This way, the packages won´t make network hops to travel between Server and Client, meaning the download time will be lower and the available bandwidth could be higher than the actual network throughput.

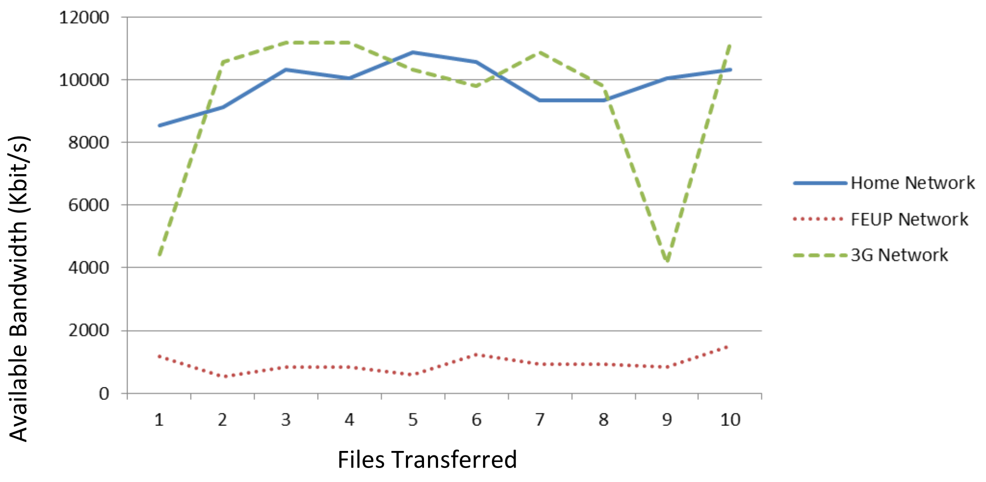


Figure 45 - Available bandwidth (Kbit/s) using the download time of a 50Kb file, transferred 10 times.

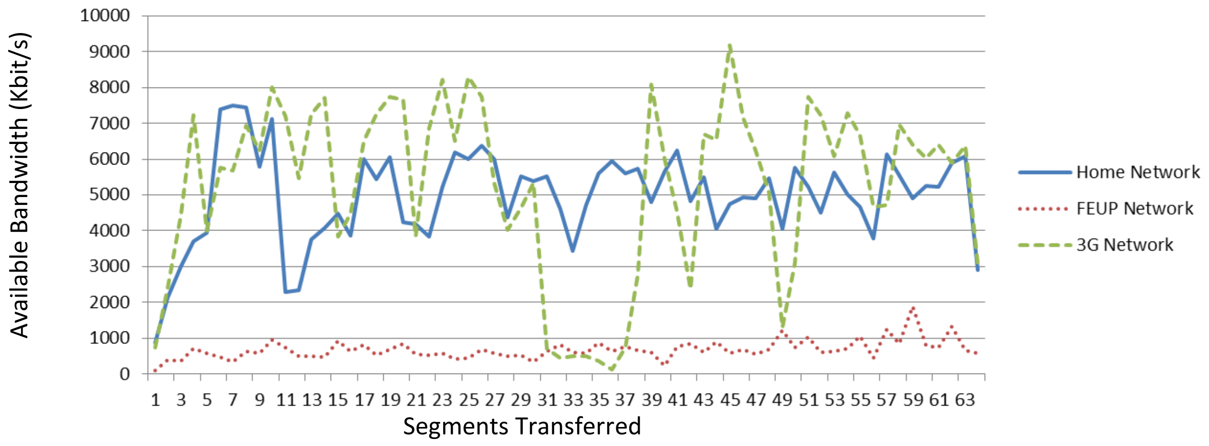


Figure 46 - Available bandwidth (Kbit/s) using the download time of a movie's segments. The movie was divided in 63 separated segments.

Analyzing the three curves, it's clear that the *3G Network*, despite being the network that reaches higher results for available bandwidth, is the one who has a more irregular behavior. This makes it the best network to simulate an environment with a lot of variation.

Table 2 - Average available bandwidth measures of different networks, using *Wireshark* and the *DASH Client* with the 50Kb file method and the segments method.

Available Bandwidth (Average)	50 Kb file (*Wireshark*)	Segments (*Wireshark*)	50 Kb file (*DASH Client*)	Segments (*DASH Client*)
Home Network	370 Kbit/s	235 Kbit/s	9852 Kbit/s	4960 Kbit/s
***FEUP* Network**	304 Kbit/s	376 Kbit/s	942 Kbit/s	683 Kbit/s
3G Network	258 Kbit/s	892 Kbit/s	9340 Kbit/s	5239 Kbit/s

Comparing the average available bandwidth of all three networks, there are a lot more available bandwidth in the *Home Network* and *3G Network*, compared to the *FEUP Network*. This proves that *FEUP Network* has a higher usage, making it the best network to simulate an environment with higher bandwidth consumption.

As for the “50 Kb file” method versus “Segments” method, the first one presents the highest results for available bandwidth. Considering that these measures have a “gain”, caused by the shortage of the download time, due to the server sharing the same network as the client, I assume that the second method’s measures are more realistic.

The *Wireshark* measurements where done in the Server side while the Client was running the tests. Comparing the *Wireshark* measures and the *DASH Client’s* measurements, the first one returns much lower results. While *DASH Clients’* measurements uses only one file exchange, between Server and Client, to calculate the available bandwidth, *Wireshark* uses the whole network traffic to do the same calculation. This explains the lower results. However it doesn’t explain irregularity of the results of the two applications. The network who has the highest/lowest measures using the *DASH Client* should be the same using *Wireshark*.

6.1.2. Noise Conditions

The noise measurements were done in different room conditions, to simulate different sound pressure levels. Two high rated applications on the *Android Market* were also used to compare the measures.

Table 3 - Sound pressure level measures of different sound sources, using *Noise Meter*, *Decibelímetro - Sound Meter* and the *DASH Client*.

Source of Sound	Sound Pressure Scale	Sound Pressure Level (*dB*)	*Noise Meter (JINASYS)* [24]	*Decibelímetro - Sound Meter (Smart Tools co.)* [25]	*DASH Client*
Hear damage (over long-term exposure)	—	85 dB	—	—	—
Traffic at a busy roadway	Very High	80 - 90 dB	58.2 dB	71 dB	69.16 dB
Passenger car	High	60 - 80 dB	50.5 dB	67 dB	60.6 dB
TV (set at home level) at 1m	Normal	55 - 65 dB	45.7 dB	58 dB	54.2 dB
Normal conversation	Low	40 - 60 dB	42.8 dB	56 dB	52.6 dB
Very calm room	Very Low	20 - 30 dB	24.9 dB	41 dB	33.4 dB
Auditory threshold at 1Khz	—	0 dB	—	—	—

To calculate the sound pressure level in *dB* it was used discrete values, calculating the average of several intercalated measures using the equation (1.1), where P is the sound pressure and $P0$ is the reference sound pressure:

$$SPL = 20\ \log_{10}\frac{P}{P0}\ , \qquad (1.1)$$

The sound pressure P is calculated using the maximum amplitude from the audio recording. The reference $P0$ equals $2 \times 10^{-5}\ Pa$, which is the auditory threshold. This value is the threshold of hearing, corresponding to a sound pressure level of 0 *dB*.

From the analysis of the measures made, the *DASH Client* return similar values when compared with the other two applications used for testing. In general the values returned represent well the sound pressure level of the respective source of sound.

The difference between the measures of the two market applications used for testing is a good prove that noise measurement using smartphone it´s not very accurate, mainly because of the algorithms used, hardware differences of the microphones and *CPU* power. The applications may compute the *FFT* of the audio file instead of using equation (1.1) or even use variations of it. Maybe the two algorithms are calibrated for different types of microphones.

Finding the sound pressure level from smartphone audio recording and change the playback volume accordingly can be tricky for three main reasons:

1. The microphones built in different devices have different characteristics and calibrations, so different devices can be more or less sensitive to sound measures, returning different values, despite the same sound pressure of the same sound source.
2. The microphones in these devices were designed for a specific frequency range, namely the voice frequency. This range occupies the spectrum from 300 Hz to 3 KHz. The recording will be limited to this range, leading to possible errors in the measurements.
3. During a video playback, a main source of noise can be the speaker of the same device, because it is so close to the microphone. So, if other noise sources in the room don't vary their noise level, the only changes that the microphone will capture are the ones from the speaker.

   This way, the system can enter an infinite loop. Because the playback volume raises if the noise level increases, if the only source of noise is the device's speaker, the microphone will detect an increase on the noise level every time the volume raises and responds to that with another volume raise. The same thing appends if the volume decreases.

To prevent this infinite loop from happen, instead of comparing a measure with the one immediately before, the application could simply take a measure and see where it fits on the sound pressure scale. According to the result, it would set the playback volume if it was too high or too low.

But, this solution has problems too, because different devices will return different results. One device can return 50 *dB* of sound pressure and other one, in the same conditions can return 80 *dB*.

6.1.3. Brightness Conditions

Ambient light level is measure in *lux* (lumen per square meter). *Lux* readings are directly proportional to the energy per square meter that is absorbed per second. Human perception of light levels is not so straightforward, because our eyes are constantly adjusting and changing that perception. However, this perception can be simplified by creating ranges of interest, with known upper and lower thresholds.

If a logarithmic scale is used on the *lux* ranges, the relation between these ranges and the associated lighting step will be roughly linear. The ranges can be normalized using equation (1.2), where x is the measurement luminance measurement in *lux*:

$$Light_{norm} = {}^{\log 10(x)}/_{5.0}\ , \qquad (1.2)$$

The *DASH Client* uses the light sensor or, if the device hasn't one, the camera to make measures of lighting levels. When using the camera, the resulting picture is analyzed on background. The perceived brightness of an image, using *RGB* color space, can be calculated using several different formulas. It was used equation (1.3) that relates the weighted distances in a *3D RGB* space, where *R*, *G* and *B* are the number of red, green and blue pixels, respectively:

$$\text{Brightness} = \sqrt{(0.241R^2 + 0.691G^2 + 0.068B^2)}\ , \qquad (1.3)$$

To normalize the value, the result is divided by 255, which is the maximum value of each color depth.

For better understanding, *RGB* color space could be imagined as a cube, where each of the three colors is an axis. In one corner there is black (*RGB* 0,0,0) and in the other corner there is white (*RGB* 255,255,255). So, if the color is close to black than it should be darker. The coefficients of each color are used to give different weight to each axis, because some colors are brighter than other.

Table 4 - Average brightness level measures of different lighting conditions, using *DASH Client* with a 3.15MP and a 5MP camera.

Lighting Condition	Lighting step	*Lux*	Light Normalized (Average)	*DASH Client* (3.15MP Camera)	*DASH Client* (5MP Camera)
Pitch Back	1	0 - 10 lux	0% - 20% (13.98%)	1.35%	1,46%
Very Dark	2	11 - 50 lux	21% - 33% (29.54%)	14.90%	15.06%
Dark Indoors	3	51 - 200 lux	34% - 46% (41.94%)	19.53%	28.48%
Dim Indoors	4	201 - 400 lux	47% - 52% (49.54%)	26.45%	32.63%
Normal Indoors	5	401 - 1000 lux	53% - 60% (56.90%)	30.98%	43,66%

Bright Indoors	6	1001 – 5000 lux	61% - 73% (69.54%)	30.45%	46.06%
Dim Outdoors	7	5001 – 10000 lux	74% - 80% (77.50%)	31.27%	48.98%
Cloudy Outdoors	8	10001 – 30000 lux	81% - 89% (86.02%)	38.05%	50.12%
Direct Sunlight	9	30001 – 100000 lux	90% - 100% (96.26%)	40.01%	52.67%

To compare and analyze the results, a *Samsung Galaxy Ace S5830* [27] was used, built in with a 5MP primary camera. The results have improved, using a 5MP camera, because the photo taken is represented with more pixels, meaning that objects on the picture are better represented and the brightness calculation will return a better value.

The overall results show that the brightness calculated is not within the range in the specified lighting condition, the measurements are too low. Although, the results are steady, changing accordingly with the lighting step.

Using the camera for measuring the lighting conditions is not an efficient process. The camera consumes a lot of resources from the device. Besides, holding and releasing the camera must be done on specific stages of the “taking picture” algorithm. If a picture is taken before the camera is released, an error will occur and the whole application will shut down. Furthermore, the measurements may vary according to hardware differences on the camera’s devices.

6.1.4. Video Playback

In this section, the usage context information methods are used during media playback, to evaluate their performance when integrated in the overall system. The media playback without using the methods mentioned before and the original media playback, using the default streaming process of *Android* were also used for performance tests.

This performance was evaluated objectively, using an option from *Android* operating system, which gives a feedback of the *CPU* usage, and subjectively, collecting the opinion of a few users, when asked to compare the different streaming sessions.

Android uses *Linux* load averages to show the user the amount of computational work that the device performs. The result is the amount of load on a *CPU* and not the load level of the *CPU* utilization. Strangely, *CPU* usage on *Android* devices is almost always near 100%. This is a good thing, because it means that all important processes are running on the background.

The load averages are represented by three numbers, representing averages over progressively longer periods of time (one, five and fifteen minutes). So, higher numbers means a problem or an overloaded device. The *CPU* load of the default device for testing before running the application was 8.52, 7.13, 3.58. *MOS* is generated by the averaging the results of people’s opinion.

Table 5 - Relationship of the MOS classification with the quality and impairment of the media content.

MOS	Quality	Impairment
5	Excellent	Imperceptible
4	Good	Perceptive but not annoying
3	Fair	Slightly annoying
2	Poor	Annoying
1	Bad	Very annoying

Table 6 - *MOS* classification and *CPU* load of different video playbacks.

Video Playback	*CPU* Load	*MOS*
Playback only	9.38, 8.42, 5.44	2
Playback + Network Conditions	10.21, 9.06, 6.88	2
Playback + Noise Conditions	10.23, 9.35, 6.22	2
Playback + Brightness Conditions	11.01, 9.56, 8.57	1
Playback + All the above	11.41, 9.97, 8.77	1

Table 7 - *MOS* classification and *CPU* load of different video streaming types.

Streaming Type	*CPU* Load	*MOS*
Original Video	9.32, 8.55, 5.54	2
Full Range - Segment: 2s	10.21, 9.06, 6.88	2
Partial Range - Segment: 2s	10.12, 9.22, 7.99	2
Full Range - Segment: 5s	11.16, 9.83, 8.37	2
Full Range - Segment: 7s	10.96, 10.24, 8.75	1
Full Range - Segment: 10s	10.35, 10.25, 8.9	1
Full Range - Segment: 12s	11.11, 10.59, 9.32	1

Analyzing the *CPU* load measures, higher results are observed, if the playback is done using the algorithms to collect the usage context information. This is caused by an increase of parallel threads running on the *CPU*. This behavior repeats itself if the streaming uses bigger segment file sizes. Because the files are bigger, the threads to request and download these files will take longer to finish. There will be a bigger accumulation of other threads, waiting to be processed, leading to an increase of *CPU* load.

About the quality of experience during the video playback, there are some considerations to be aware:

1. Synchronization of audio and video

   All different playbacks and streaming types tested had problems with synchronization of audio and video. The gap between the two will increase if the file is bigger, due to higher segment length or if the segment quality increases. To minimize this problem' effects, the best streaming type to use is with the lowest segment length. On the other hand, if the playback only contains segments with the lowest bit rate, the synchronization problems won't be as bad, but this will deteriorate the quality of experience.

A *Samsung Google Nexus S* [28], with a more recent and powerful *CPU* than the default device, encounter the same problem under the equal playback conditions. Although, the resulting gap was actually smaller than the one obtained previously.

2. Playback only versus Playback + Usage Context Information

The most important context information used for video playback adaptation is the information about network conditions. The bit rate of the segment to be requested must be smaller than the available bandwidth at that time, to avoid rebuffering. Without these measurements, the application won't be able to adapt the video playback. The other methods are less important, because they adapt properties of the device, such as volume and screen's brightness. So, they improve the quality of experience but not the quality of the video playback, because the segments downloaded will have all the same bit rate.

3. Original Video versus *DASH* Streaming

The behavior of the original video playback was expected to be this way. Because the file was encoded for progressive download to be able, the playback was a quick start. The video quality doesn't change, so if the bit rate is too high related to the available bandwidth, the playback will have a lot of glitches and rebuffering.

The *DASH* streaming fixes the glitches problem, but on the other hand, it doesn't maintain a continuous video quality. The quality may vary from being very good (2Mbps bit rate) on one second to being very poor (50Kbps bit rate) on the next one. These sudden changes occur because of the irregular behavior of the available network.

Sometimes the video playback may pause briefly, especially in the beginning, because the library assumes that the cache file being played had reached the end, even if it doesn't. The cache file was to be reopened again and start the playback in the same position where he left. Because of the synchronization problem, this pause can take a few seconds. The audio stops, but the image is still being presented for a few seconds. Then the cache file will be reopened. These few extra seconds wouldn't exist if there wasn't a synchronization problem, so the pause will behave more as a glitch. Because video is being adapted, this playback pause problem only occurs at the beginning or, if the file length is too big, one or two more times during playback.

6.2. Difficulties and Troubleshoot

During the development of this project, some difficulties were encountered. Some of them were resolved, other weren't.

1. Retrieve information from *MPD* file

The metadata file that contains all the information about the media segments is retrieved in the beginning of the playback. The information in the file needs to be easily accessed in all the classes that play a role in the segment request process.

To make this information global on the application, in the beginning a database was used. To access the information, the application needed to simply do a query to the database. These approach turn out to be inefficient, because the process was very slow. The bigger the *MPD* file, the slower the process was. The metadata can easily reach thousands of lines if there are many different representations and, if the movie file is too long, each representation would have a large number of segments.

This issue was resolved using, instead of a database, *Array Lists* and *HashMaps*, which turn out to be the best solution. *Hash Maps* are a data structure consisting of a set of keys and values in which each key is mapped to a single value. Each *MPD* element was associated with a *Hash Map* and the respective attributes were associated to the keys.

Because several elements of the same type may exist, such as <*representation*> for example, several *Hash Maps* representing that element were organized into an *Array List*. To retrieve the information, the application accesses the associated *Array List* and specifies the element and attribute.

2. Library to decode the streaming

Vitamio was the library chosen to play the video content. This library is based on *FFmpeg* and used the same way as the *Android* default video player, but this time with amplified capacities and more functionalities.

Android version >= 3.0 supports *HLS* streaming, meaning the default video player supports *.ts* files reproduction and progressive streaming. The problem is that progressive streaming has to be done manually, because the application need to control dynamically which segments have to be requested, alongside with the video presentation.

So, *Vitamio* was used to be able to do this. Some problems with *Vitamio* were:

- Finding the library

To be able to write the segments to a file and reproduce it at the same time, a lot of technologies were tested but all without success, except *Vitamio*.

The *Android* default player was limited, so an external player available in the *Android Market* would be an option. But, this way, all the functionalities of the player couldn't be fully controlled. So, there was the need to develop a video player instead of using an existing one.

FFmpeg was the next possible solution. With this library, the application would be able to manipulate the video presentation. Because *FFmpeg* is written in *C* and *C++* (native-code can be helpful so it canbe possible to reuse existing code libraries with these languages), to integrate this library in the project, *NDK* had to be installed and configured. The project would have regular java files and native code files, called from the java code, to access the library methods.

So *FFmpeg* would be an excellent solution, but very complex to build. To start, the installation and configuration of *NDK* was not trivial at all, especially since *Windows* platform were used. Second of all, it required some time studying how to manipulate the libraries methods and how to use them. *Vitamio* library was discovered a few days after configuring *NDK* to use *FFmpeg*.

Vitamio is the foundation of *VPlayer Video Player*, one of the most popular *Android* video players and itself is based on *FFmpeg*. So I could install this library, without having to install *NDK* or develop any native code, and take advantage of *FFmpeg* capacities in a very intuitive way.

- Problems with different versions of devices *CPU*

Vitamio is a great library, but the current full released version has some issues related with some *CPU* architectures, such as *ARMv6*. The default device for tests was an *ARMv6* device. So the streaming is working, but the video quality is really bad, because the audio and video are not in phase along the presentation. The audio is playing at normal speed, but the image is really slow, almost like a slow motion.

This won't be an issue if the upgraded version of the library, *Vitamio4.0*, which is not full released yet.

- Play *MP4* files

Although *MPEG-DASH* supports both *.ts* and *.mp4* files, the most common video files used are *MP4*. *MP4* is a superior format than *MPEG2-TS* [23]. According to Timothy Siglin, in two decades of existence, little has changed in that timeframe for basic *MPEG2-TS* capabilities. *MP4* files as benefits such as content/metadata separation, independent track storage, trick-play modes, backwards compatibility, seamless streaming splicing and integrated *DRM*.

The problem is that *MP4* files could not be played. The content was encoded with different parameters, but none would work. The streaming of *MP4* files require that the "*moov*" atom must be placed before the media content to be downloaded from the server first. And strangely, the same *MP4* files could be played with *Android* default player.

Maybe when *FFmpeg* encodes the files, it places this atom at the end of file, and for this reason, not suitable for streaming. Maybe the parameters used for encoding are not supported by *Android*.

6.3. Future Work

Several suggestions of possible expansions to the work here described are presented:

1. Live streaming

The developed *DASH* client doesn't support live streaming. The *DASH* protocol allows this, using a live profile. The system developed would need to support:

- *MPD* parsing for live content on the *Client* side. The algorithm developed to parse the *MPD* file and to request the segments are not suited for live content.
- Live content generation on the *Server* side. It is required a video recorder device connected to the Server. The Server must be able to receive the content and encode it right away, to be available to send to the client when requested.

2. Other platforms

The *DASH* client was developed only as an *Android* application. Versions for other operating systems, such as *Apple – IOS, Microsoft – Windows Mobile, Nokia – Symbian* and *Black Berry – RIM* can be a possibility in the future.

Another important platform would be a web based application. The recent *HTML5* protocol allows media reproduction. This feature would be used to display the media files, controlled using *JavaScript*. In fact, *DASH* is being developed with the intent to be used with *HTML5* media reproduction capacity. This way, the technology will go global, targeting much more users, because they only need a browser that supports *DASH*. Besides, they can use it in any machine with Internet connection, not just their phone.

3. *Vitamio*'s library update

The *DASH Client* uses *Vitamio3.0* to reproduce video content. This version has known problems with different *CPUs*. There are a lag between audio and image display.

Vitamio4.0 is expected to be fully released later this year. This version would not have any problem with different *CPU* devices and will support many new features that can be used for a better user experience.

4. Context Information

New usage context information sources could be used to give a better user experience:

- Object and Face Detection

The *DASH* client supports face detection, but it really doesn't do anything with that information. This can be used to detect if there is more than one person in the room. Maybe the user doesn't want other people to watch what he is watching for some reason, so privacy issues must be considered. In fact, in the future the application could use face detection together with face recognition, displaying the content only for the intended user.

- *CPU* Usage

More resources used to collect the usage context information means more *CPU* usage. The application would run more threads in parallel and this leads to more *CPU* memory consumption.

This *CPU* usage can be an issue, because if the memory consumption is too high, video playback can be compromised. So, measure *CPU* memory usage can be an important feature.

If the memory consumption is too high, the application would kill threads used to control the camera or the microphone. If that wasn't enough, the application would request segments with a lower frame rate, meaning fewer frames per second to be processed by the *CPU*.

6.4. Revision of work

The project developed consists on a prototype for a web video streaming service such as *VoD*. This prototype is used to study different solutions to increase the quality of experience of the user.

The system developed is constituted by a web *Server*, the service provider and a smartphone application to request the service. The web Server stores the files to be delivered after their request by the user. The smartphone application is for *Android* devices. It functions like a video player, using a streaming protocol, *MPEG-DASH*, to adapt the video content.

To display the video, the application follows a progressive streaming strategy, using *Vitamio* library, to be able to control which content form should be requested next.

To adapt the video content, the application uses different strategies to decide the best content form to deliver. For this it uses context information collected from *Android* sensors and resources. It uses available bandwidth to decide the quality of the content to be requested; noise conditions, using the microphone measures, to decide the volume level of playback; brightness conditions, using the light sensor or camera measures, to decide the brightness level of the screen; phone position, using the orientation sensor, to match the video resolution with the screen resolution; face detection, using the camera, to make a decision if there are privacy issues.

The application does what it supposes to do. It adapts the video using the available bandwidth measures, avoiding the constant glitches observed on the original video playback. The volume and screen brightness adaptation works well, although it can become annoying, if some measures doesn´t reflect correctly the real conditions, because of the different devices used.

The subjective evaluations don’t correspond to the objective ones, because of the secondary effect of the overall functionalities. I’m talking mainly of the synchronization problem on the video playback. This results on a poor quality of experience, which is the opposite of what was expected to reach with this project. But, between the playback of the original file and the *DASH* streaming, although it is bad, the quality of experience actually is better with the second one.

References

[1] "Investigation Report on Universal Multimedia Access", Eiji Kasutani, January 2004.

[2] Composite Capabilities/Preference Profiles. Available on http://www.w3.org/Mobile/CCPP/. Access on January 2013.

[3] "A Survey on Delivery Context Description Formats – A Comparison and Mapping Model", Christian Timmerer, Johannes Jabornig, Hermann Hellwagner, Department of Information Technology (ITEC), Klagenfurt University Austria, February 2010.

[4] "Evaluation of Usage Environment Description Tools", Robbie De Sutter, Frederik De Keukelaere, Rik Van de Walle.

[5] "Experiences in Using CC/PP in Context-Aware Systems", Jadwiga Indulska, Ricky Robinson, Andry Rakotonirainy, Karen Henricksen.

[6] HTTP Live Streaming Overview. Available on http://developer.apple.com/library/mac/#documentation/NetworkingInternet/Conceptual/StreamingMediaGuide/Introduction/Introduction.html. Access on October 2012.

[7] IIS Smooth Streaming Technical Overview. Available on http://www.microsoft.com/en-us/download/confirmation.aspx?id=17678. Access on October 2012

[8] HTTP Dynamic Streaming on the Adobe Flash Platform. Available on http://www.adobe.com/products/httpdynamicstreaming/pdfs/httpdynamicstreaming_wp_ue.pdf . Access on October 2012.

[9] The Java Tutorials. Available on http://docs.oracle.com/javase/tutorial/. Access on April 2013.

[10] Eclipsepedia. Available on http://wiki.eclipse.org/Main_Page. Access on April 2013.

[11] Android Developers. Available on http://developer.android.com/index.html. Access on April 2013.

[12] Vitamio. Available on http://www.vitamio.org/en/. Access on March 2013.

[13] "The MPEG-DASH Standard for Multimedia Streaming Over the Internet", Anthony Vetro, 2011.

[14] "Adaptive Streaming of Audiovisual Content using MPEG DASH", Truong Cong Thang, Quang-Dung Ho, Jung Won Kang, Anh T. Pham, 2012.

[15] "A Test-Bed for the Dynamic Adaptive Streaming over HTTP featuring Session Mobility", Christopher Muller, Christian Timmerer.

[16] "Guidelines for Implementation DASH264 Interoperability Points", DASH Industry Forum, January 2013.

[17] "Implementation of DTS Audio in Dynamic Adaptive Streaming over HTTP (DASH)", dts, 2012.

[18] ISO/IEC 23009-1, "Information technology – Dynamic adaptive streaming over HTTP (DASH)", 2012.

[19] FFmpeg Documentation. Available on http://ffmpeg.org/documentation.html. Access on May 2013.

[20] MP4Box General Documentation. Available on http://gpac.wp.mines-telecom.fr/mp4box/mp4box-documentation/. Access on May 2013.

[21] Apache Tomcat. Available on http://tomcat.apache.org/. Access on March 2013.

[22] Servlet Essentials. Available on http://www.novocode.com/doc/servlet-essentials/. Access on March 2013.
[23] "Unifying Global Video Strategies: MP4 File Fragmentation For Broadcast, Mobile and Web Delivery", Timothy Siglin, 2011.
[24] Noise Meter - JINASYS. Available on https://play.google.com/store/apps/details?id=com.pjw.noisemeter&feature=search_result#?t=W251bGwsMSwyLDEsImNvbS5wancubm9pc2VtZXRlciJd. Access on June 2013.
[25] Decibelímetro – Sound Meter - Smart Tools co. Available on https://play.google.com/store/apps/details?id=kr.sira.sound&feature=search_result#?t=W251bGwsMSwyLDEsImtyLnNpcmEuc291bmQiXQ. Access on June 2013.
[26] LG Optimus One. Available on http://www.gsmarena.com/lg_optimus_one_p500-3516.php. Access on June 2013.
[27] Samsung Galaxy Ace S5830. Available on http://www.gsmarena.com/samsung_galaxy_ace_s5830-3724.php. Access on June 2013. Access on June 2013.
[28] Samsung Google Nexus S. Available on http://www.gsmarena.com/samsung_google_nexus_s-3620.php. Access on June 2013.
[29] Android Developers Dashboards – Platform Versions. Available on http://developer.android.com/about/dashboards/index.html. Access on June 2013.