

POLITECNICO DI TORINO

III Facoltà di Ingegneria

Laurea Specialistica in Ingegneria delle Telecomunicazioni



**Development and Implementation of an
Adaptive HTTP Streaming Framework for
H.264/MVC Coded Media**

Adviser:

Prof. Gabriella OLMO – Politecnico di Torino

Co-Adviser:

Ing. Emanuele QUACCHIO – STMicroelectronics

Candidate:

Juan Esteban HURTADO GUZMÁN

November 2010

Acknowledgments

I would like to express my deepest gratitude to Emanuele Quacchio, my supervisor at STMicroelectronics, for the patient guidance and all the intuitions provided. Without his expert support, carrying through this study would have been much harder and taken longer. I'm indebted to him for effective team work, and encouraging through the whole process. I would also like to thank Prof.ssa Gabriella Olmo, for providing this thesis opportunity and constant guidance.

Many thanks to all my friends who have been around all these years, who have been helping me at different occasions and in one way or another have influenced this thesis

I must acknowledge my girlfriend and best friend, Paula, without her love, patient, encouragement and editing assistance, I would not have finished this thesis.

Lately, and most importantly I wish to thanks my family. Victoria Eugenia, Juan Martin and Alejandro. They have always supported and encourage me to do my best in all matters of life. To them I dedicate this thesis.

Abstract

3-D/MVC video streaming refers to the continuous, interactive and real-time delivery of 3-D/Multi-view media content over networks that allows user interactions without a prior download. The importance of streaming technologies can be easily explained by some facts that are happening around us: The number of internet users, which this year is expected to reach the incredible number of 2000 million (1); the diversity of multimedia enabled devices with better processing power, improved user interfaces, lower prices and expanded services; the number of internet connected terminals is expected to be two or three times the number of internet users (2); the present deployment of better but every time more diversified access technologies. The fact that 64% of the internet traffic is video (2) (user-generated content or professional content), and the consumer expectation of access services 24x7, no matter the device, make the streaming technologies and the efficient delivery of contents a critical and crucial aspect in the whole scenario. A deeper discussion about these motivations will be held in chapter one.

On the other hand, 3-D content on Internet has not yet become popular due to several reasons, such as the lack of technology in the past to run, deploy and transport the content, its typically large volume and the limited network bandwidth. The heavy data and processing requirements of 3-D streaming challenges the scalability of client-server delivery methods. As technologies becomes available, 3-D content is becoming really popular and so-it will start to be requested as any other video content, and even more. However, streaming and deliver of 3-D/multi-view video requires especial interactions between the user and media server, yet more, a good streaming framework should also look for adaptivity, compatibility, etc.

This final thesis work involved the development of an adaptive HTTP streaming model with an efficient signaling technique for Multi-view video. Some of the features implemented were: on the fly controllability by the server in order to adapt the stream to the server capabilities (adaptivity at server side) or type of user, on the fly traffic redistribution to enhance QoS and network resources (adaptivity at network side), and finally, enrichment of services/quality on the fly, that allows users to select the preferred 2D/3-D view-points and even let the client automatically choose the best stream to fit its requirements on energy

consumption, resolution and band (adaptivity at client side). The main reason to use a standard web protocol such as HTTP for MVC content is the “transparency” of it across Firewalls and proxy-servers and the compatibility with almost any OS distribution, letting the stream reach a wide variety of devices across a wide variety of networks.

For flexibility and compatibility reasons the MVC server (the streaming server) was implemented using the concept of CGI (common gateway interface) which is a standard that defines how a web server software could delegate the generation of dynamic content to a third application. All major commercial and open source HTTP servers like Apache or Null allow CGI. The HTTP streaming client was implemented inside VLC player, a free and open source media player and multimedia framework written by the VideoLAN project (3).

This thesis is divided in eight chapters. The first chapter is an introduction aimed to explain the importance of media delivery technologies as well as some of the motivations that led this project. The second chapter is a brief description of today’s 3-D/Multi-View video coding technologies and standards. Chapter three introduces the most important media streaming technologies and standards. The fourth chapter describes the whole adaptive http streaming framework for H.264/MVC coded media proposal. Chapter five presents some proposals to achieve better performance with the streaming framework, most of them related to efficient MVC coding structures for the adaptive HTTP streaming framework. The sixth chapter explain the entire framework implementation. Chapter seven show significant tests and results and the eighth chapter propose some future works.

The thesis work was developed at the Advance System Technology Central Research and Development division at STMicroelectronics Srl, Milan. The implemented work is part of a project funded by the European Community (COAST, Content Aware Searching, retrieval and streaming), in which STM is partner. COAST project has its main focus on the search and delivery across heterogeneous networks of video content encoded using the most modern video compression technologies (SVC/MVC). In particular lot of emphasis is given to 3-D/Multi-View streaming of video contents in high definition. The implemented solution has been verified and compared with efforts in the MPEG standardization committees and with important international partners of ST (mainly Heirtz Heinrich-Institute of Berlin).

Table of Contents

Acknowledgments.....	ii
Abstract	iii
Table of figures	viii
Tables and calls	ix
1 Introduction.....	1
1.1 State of the internet and the importance of streaming technologies.....	2
1.2 Streaming (vs. download once and own)	3
1.3 The importance of adaptivity.....	5
1.4 Brief introduction to HTTP as a media delivery protocol	6
1.5 Streaming 3-D/Multi-view content.....	8
2 3-D/Multi-View Video Coding Technologies and Standards.....	9
2.1 Standard H.264/AVC	11
2.1.1 High-level structure of an H.264/AVC bit-stream.....	11
2.1.2 The video coding layer (VCL)	13
2.2 Standard H.264/MVC.....	15
2.2.1 MVC Stream data chain.....	16
2.2.2 H.264/MVC coding structure is adaptive friendly	16
3 Media Streaming Technologies and Standards	18
3.1 Live streaming and Video on Demand.....	19
3.1.1 Video on Demand and Unicast connections	19
3.1.2 Live streaming and Multicast connections.....	20
3.1.3 The hybrid solution	21
3.2 Components of a streaming framework	21
3.2.1 Media coding format.....	22
3.2.2 Streaming server.....	22
3.2.3 Control mechanisms.....	22
3.2.4 Buffering schemes.....	24
3.2.5 Streaming Client application.....	25

3.3	Protocols for media streaming.....	26
3.3.1	Internet protocol IP	26
3.3.2	TCP and UDP protocols.....	26
3.3.3	RTP/RTSP/SDP protocols	27
3.3.4	HTTP/1.1 protocol	28
3.4	HTTP streaming	30
3.4.1	HTTP progressive download.....	30
3.4.2	HTTP adaptive bitrate streaming	31
3.4.3	HTTP streaming. SoA in commercial solutions	33
3.5	HTTP streaming. 3GPP's Standard	34
3.6	The Media presentation description file (MPD).....	34
3.6.1	Period	36
3.6.2	Representation.....	36
3.6.3	Segments	36
3.6.4	Media Presentation Description Updates	37
4	Description of the Adaptive HTTP Streaming Framework for H.264/MVC Coded Media	38
4.1	Framework overview.....	39
4.2	The manifest file or Media Presentation Description file	41
4.2.1	MPD for H.264/MVC contents	42
4.2.2	MPD profiles.....	44
4.3	Framework architecture.....	45
4.4	The framework's streaming process.....	46
4.4.1	Requesting a media resource.....	47
4.4.2	Parsing MPD file at client side and start requesting segments	47
4.4.3	Parsing HTTP requests at server side.....	48
4.4.4	Segments requests and delivery with HTTP/1.1	50
4.4.5	MPD updates	50
4.5	Scalability and adaptivity features.....	51
4.5.1	On the fly adaptation at server side.....	51

4.5.2	On the fly traffic redistribution	52
4.5.3	Playback tricks and preferred 2D/3-D view-points.....	53
4.5.4	Automatic client adaptation	54
5	Efficient MVC Coding Structures for the Adaptive HTTP Streaming Framework	55
5.1	Efficient coding structures for bandwidth saving.....	56
6	Framework Implementation	59
6.1	Media Resources	60
6.1.1	Creating H.264/MVC streams with JMVC 8.0	61
6.1.2	Generation of segments.....	63
6.2	Implementation of an adaptive HTTP streaming enabled server	63
6.2.1	The HTTP server: NULL HTTPD	64
6.2.2	Common Gateway Interface (CGI)	65
6.2.3	The adaptive streaming extension for HTTP server	67
6.2.4	Control mechanisms by third party applications.....	68
6.3	Implementing the streaming framework at client side	69
6.3.1	VLC 0.8.6.....	71
6.3.2	Decoder Plug-In To Interface VLC With JMVC 8.0	73
6.3.3	Custom Demuxer	73
6.3.4	Custom HTTP Client.....	74
7	Test Bed, Tests and Results	78
7.1	Sequence for the test bed: Nurburgring race 3D	79
7.2	Setting up server	84
7.3	Requesting a media resource with VLC	85
7.4	Test: Traffic Redistribution and Automatic Adaptation.....	85
7.5	Test: Playback tricks and preferred 2-D/3D view points	90
8	Future Work	91
8.1	Free View-Point streaming.....	92
8.2	Broadcast scenario.....	94
	Bibliography.....	95

Appendix A: Troubleshooting for the client's compilation process	98
--	----

Table of figures

Fig 1 New Computing Cycle (2)	2
Fig 2 Video rapid growth in mobile internet traffic	3
Fig 3 Media content and cloud computing	4
Fig 4 Home networks will be heterogeneous networks	5
Fig 5 requirments on QoS for diferent content providers	6
Fig 6 broadcast vs. on demand	7
Fig 7 Multi-camera setup for 3D acquisition and captured multi-view video.	8
Fig 8 3-D/multi-view video especial interactions	8
Fig 9 Video bitrate calculation for a typical HD sequence	10
Fig 10 High-level strucutre of H.264/AVC bitstream.....	12
Fig 11 A picture partitioned in slices	13
Fig 12 Interprediction between I frames , P frames, and B frames.....	13
Fig 13 Basic coding structure of H.264/AVC for a macroblock.....	14
Fig 14 Typical Multi-view coding structure for 5 Views	15
Fig 15 MVC stream data chain	16
Fig 16 operations points for the coding structure in Fig 14	17
Fig 17 Unicast vs. Multicast connection in terms of bandwidth use (14).....	20
Fig 18 Architecture of a Multimedia Streaming system	21
Fig 19 streaming client can be a module implemented inside a media player	25
Fig 20 Protocol stack for RTSP/RTP streaming	28
Fig 21 HTTP adaptive streaming	32
Fig 22 System architecture for Adaptive HTTP streaming.....	34
Fig 23 MPD structure proposed by 3GPP	35
Fig 24 MPD xml example with one representation	43
Fig 25 MPD xml example with four representations	44
Fig 26 Different MPD profiles for the same media resource.....	45
Fig 27 Framework architecture	46
Fig 28 At the begining of the session, server sends the MPD file	47
Fig 29 Parsing MPD at client side.....	48
Fig 30 Parsing HTTP request at server side.....	49
Fig 31 MVC Data Chain Filtering	49
Fig 32 two diferent schemes for request of segments with HTTP/1.1	50
Fig 33 On the fly adaptation at server side	52

Fig 34 traffic redistribution between CDN servers	53
Fig 35 Proposed multiview coding structure	56
Fig 36 Op Points and Possible Enrichments for the Enhanced Structure	57
Fig 37 MVC stream Data Chain filtered in real time.....	57
Fig 38 Comparison between coding methods.....	58
Fig 39 Creating H.264/MVC streams with JMVC 8.0.....	62
Fig 40 Server side architecture.....	64
Fig 41 Common gateway interface	66
Fig 42 CGI communication with http server.....	66
Fig 43 adaptive streamig application flow chart for each segment request	68
Fig 44 Script for update the MPD at server side.....	69
Fig 45 flow of data in VLC	70
Fig 46 flow chart for methods Open() and Connect().....	75
Fig 47 Flow chart for method Request().....	76
Fig 48 Flow chart for method MPD()	76
Fig 49 Flow chart for method Read().....	77
Fig 50 Test bed architecture	79
Fig 51 Nurburgring race main MPD profile	82
Fig 52 Nurburgring race traffic redistribution MPD profile	83
Fig 53 Nurburgring race adaptation MPD profile.....	83
Fig 54 Editing bandwidth shaper parameters	84
Fig 55 Test bed for the First Test	86
Fig 56 HTTP requests flow graph.....	86
Fig 57 VLC analyze MPD and generate playlist with 2D and 3D segments	87
Fig 58 VLC requests the first media segment (2-D representation)	87
Fig 59 Server 2 has more bandwidth.....	88
Fig 60 VLC displaying a 3D representation	88
Fig 61 Traffic reallocation to enhance QoS	89
Fig 62 client's adaptation process after a traffic reallocation at CDN side	89
Fig 63 HTTP requests flow graph Test 2	90
Fig 64 TCP flow between server and client. Test2	90
Fig 65 Various FTV system architectures.....	93
Fig 66 FVV streaming.....	93

Tables and calls

TABLE 1 Overview of major HTTP adaptive streaming systems	33
TABLE 2 HTTP Adaptive framework overview	41

TABLE 3 Executables provided by the JMVC software	61
TABLE 4 Configuration settings for Null httpd	64
TABLE 5 Conciliated parametes for the buildall.sh script	71
TABLE 6 VLC source tree	72
TABLE 7 VLC modules directory	72
TABLE 8 important structures and methods of the decoder plug-in	73
TABLE 9 important structures and methods of the custom demuxer.....	74
TABLE 10 important methods of the custom HTTP client	75
TABLE 11 Encoder configuration file for nurburgring sequence	81
TABLE 12 MPD profiles for the sequence nurburgring race	81
TABLE 13 Parameters passed to VLC with the launch call.....	85
CALL 1 Basic jmvc encoder call	62
CALL 2 Basic jmvc BitStream Assembler call	62
CALL 3 Call for the Extraction of a specified operation point.....	63
CALL 4 URL to request a cgi service.....	66
CALL 5 URL to request a segment to the adaptive streaming extension.....	67
CALL 6 Launch Null http server	84
CALL 7 Launch bandwidth shaper	84
CALL 8 request the Nurburgring media resource with VLC	85

1

Introduction

3-D/MVC video streaming refers to the continuous, interactive and real-time delivery of 3-D/Multi-view media content over networks that allow user interactions without a prior download. The importance of development and deployment of adaptive streaming technologies can be easily explained by today's state of internet. Incorporate 3-D or Multi-view concepts to streaming technologies is becoming crucial because of the importance of 3-D video and interactive video in today's customers, markets and technology trends. The objective of this chapter is to leave settle and clear the importance of the words adaptive, streaming, HTTP, 3-D/MVC to understand the whole of "Adaptive HTTP Streaming Framework for H.264/MVC Coded Media".

1.1 State of the internet and the importance of streaming technologies.

Although the number of internet users went beyond 1.9 Billion (1) the usage growth remains robust with an average +13% Y/Y growing. Internet is entering into a new computing cycle, the "mobile internet" cycle where devices other than desktop computers are accessing and offering internet services [Fig 1]. Thanks to the reduced usage frictions via better processing power, improved user interfaces, smaller form factors, lower prices and expanded services. The number of devices is expected to be two or three times the number of internet users (2). The devices' zoo is constituted of more than just Smartphone: Kindle, tablet pc, PDA, car electronics, GPS, mobile video, home entertainment, games, home appliances, etc.

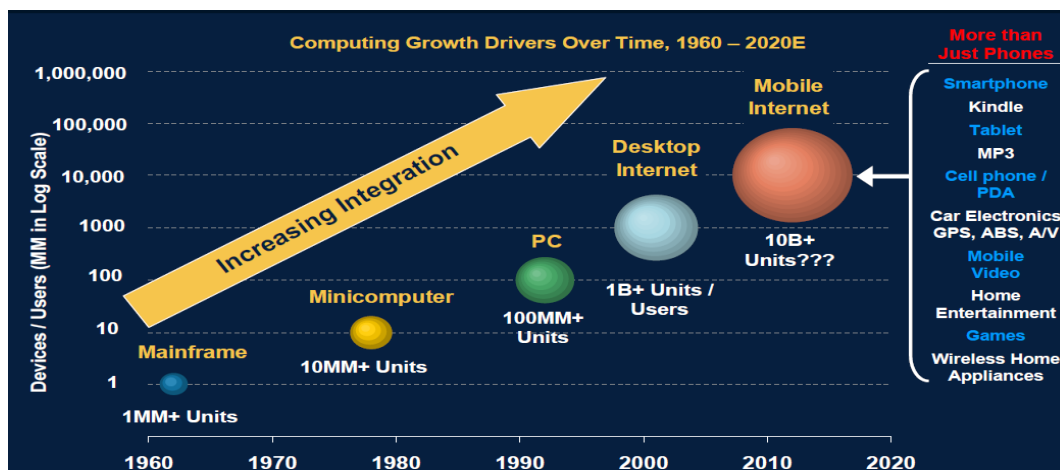


FIG 1 NEW COMPUTING CYCLE (2)

It can also be added to this scenario, today's deployment of better and every time more diversified access technologies (deep optic fiber access, Wi-Fi, 3G, etc); therefore, it is not eccentric to expect consumers wanting to find, select, watch media content specially video whenever and wherever they want, these trends are driving the demand for more and more interactive and online applications. The fact that 64% of the internet traffic is video [Fig 2] (user-generated content or professional content), tell us then how important is to have open and flexible multimedia stacks, efficient networking functions and efficient video delivery implementations and here is where streaming technologies become critical and play an important role inside this whole scenario, to efficiently transport all this content and offer a better user experience.

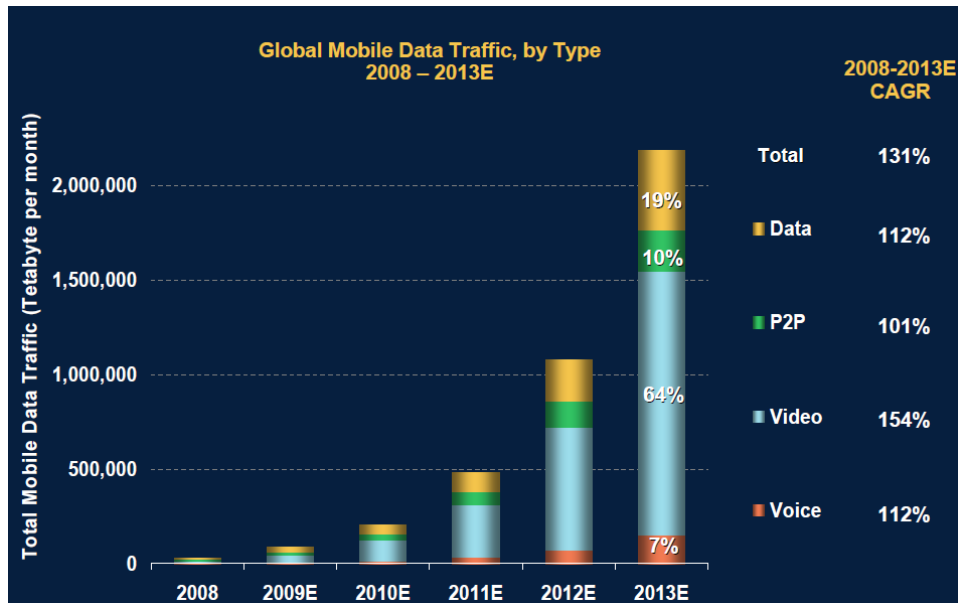


FIG 2 VIDEO RAPID GROWTH IN MOBILE INTERNET TRAFFIC

1.2 Streaming (vs. download once and own)

There are many reasons that explain why streaming is gaining ground over the old paradigm of download once and own (and not just only the obvious one of memory space requirements); we will mention two in this work, cloud computing and adaptivity. The first one is the new paradigm shift into cloud computing or Internet-based computing, whereby shared resources, software, and information are provided to computers and other devices on demand (4). We are facing

consumers that expect to get their stuff 24x7 from palms of their hands no matter if the device is a desktop pc or other mobile internet device. Cloud computing offers many key features that assure success of this new paradigm by offering remarkable advantages to users and service providers. For example better and easier user interfaces, remote processing of information in the cloud and not on the client side, device and location independence that enables users to access systems using a web browser regardless of their location or what device they are using, scalability, reliability, security, etc (5) (6). With all this in mind it becomes clear that connectivity is one of the foundations of cloud computing, and even more for media providers in the cloud [Fig 3]; streaming technologies find a place on these commitments, offering reliable, adaptable, scalable delivery of the content.

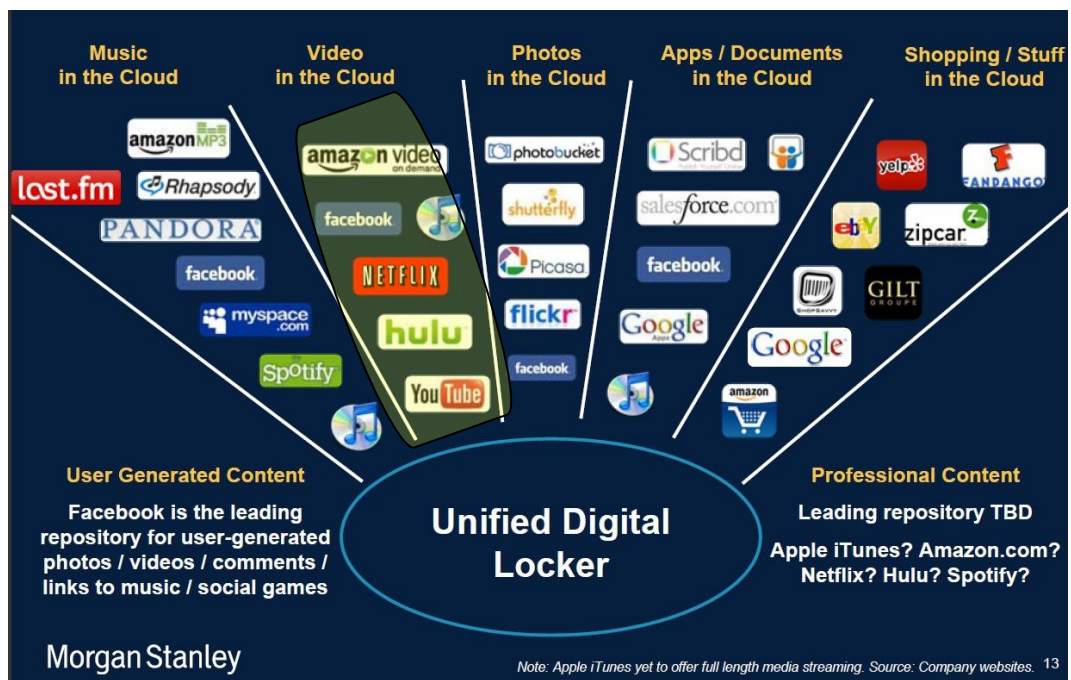


FIG 3 MEDIA CONTENT AND CLOUD COMPUTING

The second reason that explains why streaming is gaining ground over the old paradigm of download once and own is adaptivity, which explained in next section.

1.3 The importance of adaptivity

Adaptivity refers to the capacity of adapting a stream to match with heterogeneous network conditions, different type of clients or target device capabilities. It is an important property for today's internet paradigm for all the reasons mentioned throughout this chapter. For example the future home networks will be heterogeneous networks (Wi-Fi, Power-line modem) composed off heterogeneous terminals (PCs, Set Top Boxes, TVs, Tablets, Smartphones) [Fig 4], there is an implicit need to adapt the stream to the internal network conditions and terminals; e.g. television could play full HD 3-D streams with many view points while smartphones could only play 2-D sequences at lower resolutions. Another interesting scenario could be, for example, when there is a television at home playing HD 3-D content with eight possible view-points and then other television requesting a different media; however, the first one may be actually using 80 % percent of the total bandwidth, with no room for other streams. The solution could be to adapt both streams to offer, for example, just one 3-D view-point at HD, letting bandwidth to stream other media.

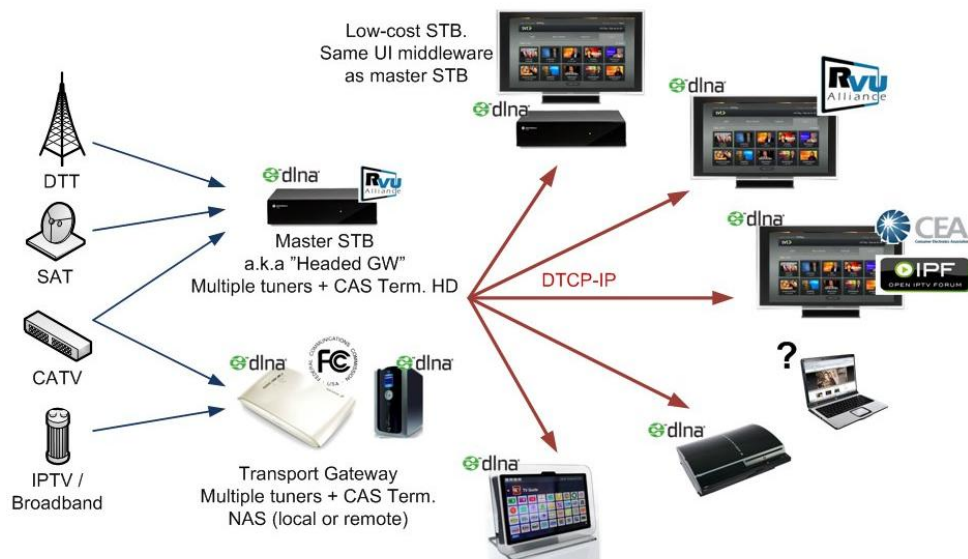


FIG 4 HOME NETWORKS WILL BE HETEROGENEOUS NETWORKS

The concept of adaptivity could be extended to the media server side or to content delivery network side, and that is one of the most important features implemented

in this thesis work. Content providers are likely to require varying levels of assured QoS [Fig 5]. For example, a content provider could have two categories to differentiate clients: premium clients and base clients. Premium clients paid for the service; therefore they expect to have always the best possible quality. The content provider should be able to adapt the stream to the user category, richer streams to premium users and lighter streams to base users. Imagine a situation where a content provider is streaming a live event offering 3-D eight view points to premium users and 3-D one view-point with advertisings to base users: and at some time many base users connect to the content provider requesting the event. Such situation jeopardizes the premium users with possible lags; the content provider should be able to adapt on the fly the stream of the base users to an even lighter stream to assure QoS of premium users. Another “adaptivity” solution could be to redistribute on the fly traffic between media servers, relocating users in other servers with reduced load.

CONTENT PROVIDERS LIKELY TO REQUIRE VARYING LEVELS OF ASSURED QoS

	Examples	Requirement for Assured QoS
Paid for programming	- Catch-up - Movies/series - Other paid-for content 	
Free VoD/catch-up	- Catch-up - Movies/series - Clips 	
Lower quality streaming	- Streamed & peer-to-peer music services - User-generated video content 	
Other	- Website content - Games - Advertising 	

FIG 5 REQUIRMENTS ON QOS FOR DIFERENT CONTENT PROVIDERS

1.4 Brief introduction to HTTP as a media delivery protocol

One of the trends that have emerged in the streaming media industry in recent years is a change from the classical streaming protocols (RTSP, MMS, RTMP, etc.) into plain HTTP download. There are a lot of examples of this tendency

today (YouTube, dailyMotion, etc) that just use primitive progressive download to deliver their content. There are several strong reasons for this industry trend (7):

- Web download services have traditionally been less expensive than media streaming services offered by CDNs and hosting providers.
- Media protocols often have difficulty getting around firewalls and routers because they are commonly based on UDP sockets over unusual port numbers. HTTP-based media delivery has no such problems because firewalls and routers know to pass HTTP downloads through port 80.
- HTTP media delivery doesn't require special proxies or caches. A media file is just like any other file to a Web cache.
- It is much easier and cheaper to move HTTP data to the edge of the network, closer to users.

Even though streaming protocols are designed with media delivery in mind, the fact of the matter is that the Internet was built on HTTP and optimized for HTTP delivery. Consequently, this begged the question: "Why not adapt media delivery to the Internet instead of trying to adapt the entire Internet to streaming protocols?" (7). Move Networks[®] proved on several occasions in 2008 that HTTP-based media delivery could be done successfully on a large scale (7) both on-demand and live. Even if people think HTTP is better suited for on demand scenarios, it could behave very well on broadcast scenarios if proper intelligence and web caches are present. Anyway, video as Unicast stream is expected to be multiplied by 3 in two years [Fig 6].

Toward more delinearisation, video as unicast stream will be multiplied by 3 in two years

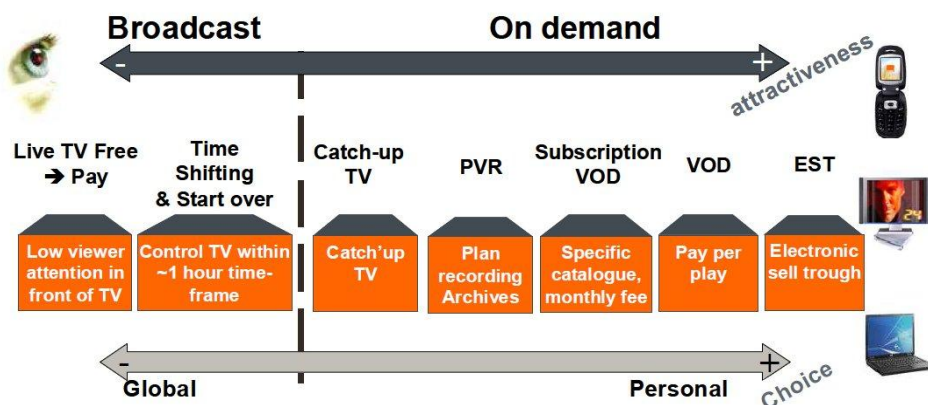


FIG 6 BROADCAST VS. ON DEMAND

1.5 Streaming 3-D/Multi-view content

3-D content on Internet has not yet become popular due to several reasons like the lack of technology in the past to run, deploy, and transport the content, its typically large volume and the limited network bandwidth. The heavy data and processing requirements of 3-D streaming challenges the scalability of client-server delivery methods. As technologies become available, 3-D content is becoming really popular and so it will start to be requested as any other video content and even more. Among 3D video services, the multi-view is a newly emerging, area that can provide a variety of media services. Multi-view video consists of video sequences that are captured simultaneously from a set of cameras with different view-points. The multi-view video can be presented with a monoscopic, a stereoscopic or a multiscopic video according to viewer's display type. Therefore, viewers can enjoy not only their desired view, but also special visual effects. However streaming of 3-D/multi-view video requires especial interactions (8) [Fig 8] between the user an media server, and a good 3-D streaming framework should efficiently save bandwidth when possible, as explained later in chapter 2 and chapter 5. It is easy to realize the minimum new special interactions needed in the playback of 3-D/Multi-view media, for example, 2-D to 3-D switching and change of view-point or angle of view [Fig 7].

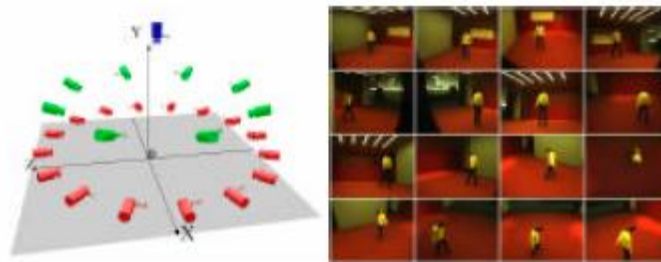


FIG 7 MULTI-CAMERA SETUP FOR 3D ACQUISITION AND CAPTURED MULTI-VIEW VIDEO.



FIG 8 3-D/MULTI-VIEW VIDEO ESPECIAL INTERACTIONS

2

3-D/Multi-View Video Coding Technologies and Standards

When a video signal is converted into a digital signal it uses large amounts of data, so large that it is beyond the capacity of any practical existing distribution and storage system. The compression process is intended to provide a digital compact representation of the signal. In general the goal is to minimize the bit-rate to fit the limited capacity of the transmission channel or storage hardware. The size and bitrate of an uncompressed sequence recorded in HD with color model YUV 4:2:2 at 30 FPS is show in [Fig 9].

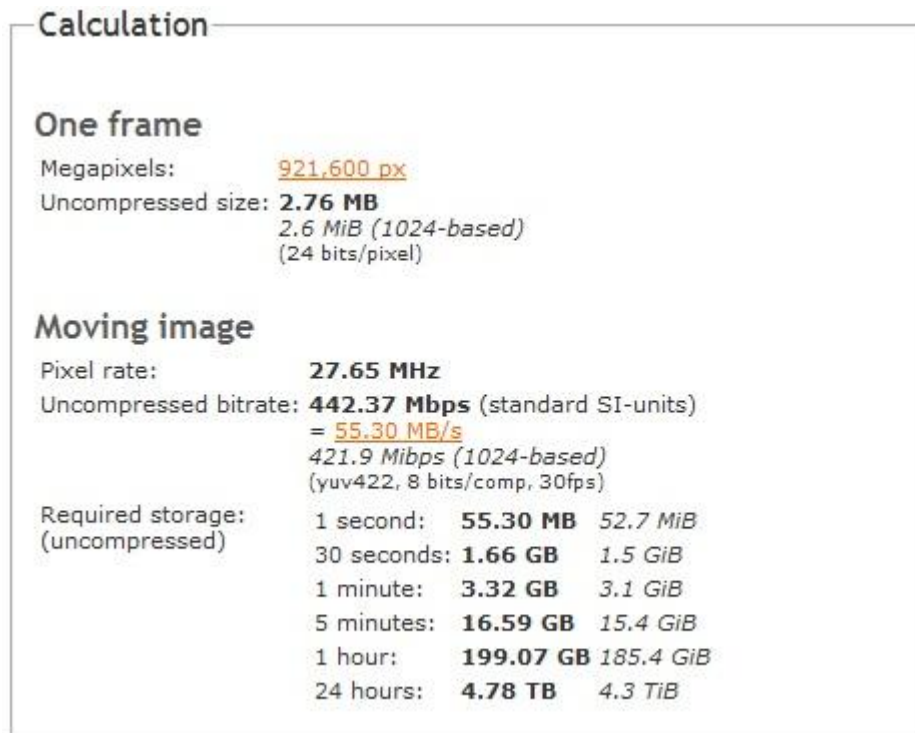


FIG 9 VIDEO BITRATE CALCULATION FOR A TYPICAL HD SEQUENCE

Due to video compression the same sequence can be transmitted with a bitstream up to a few Kbps. Most video compression is lossy, meaning this that it operates on the premise that much of the data present before compression is not necessary for achieving good perceptual quality. There are many coding technologies for video compression, we will focus on H.264/AVC (9), an industry leading standard for video compression and in H.264/MVC, an extension of H.264/AVC used to efficiently code and compress video sequences that are captured simultaneously from a set of cameras with different *viewpoints* or views of a scene. The adaptive HTTP streaming framework presented in this thesis uses H.264/MVC as a coding technology for the video intended to be delivered.

2.1 Standard H.264/AVC

H.264/AVC also known as MPEG-4 Part 10 is an industry leading standard for video compression. Developed by the ITU-T Video Coding Experts Group (VCEG) together with the ISO/IEC Moving Picture Experts Group (MPEG), it was the product of a partnership effort known as the Joint Video Team (JVT). H.264 is an open licensed standard that supports the most efficient video compression techniques available today and provides enough flexibility to allow the standard to be applied to a wide variety of applications on a wide variety of networks and systems. It covers all forms of digital compressed video: from low bit-rate Internet streaming applications to HDTV broadcast and Digital Cinema applications with nearly lossless coding. Because H.264 encoding and decoding requires significant computing power, software implementations that run on a general-purpose CPU are typically slow, especially when dealing with HD video content. To reduce CPU usage or to do real-time encoding, special-purpose hardware may be employed, either for the complete encoding or decoding process, or for acceleration assistance within a CPU-controlled environment.

It is not inside the scope of this thesis work to explain the technical facts about H.264, instead a brief introduction to the coding process and the final bit-stream structure of H.264 that make it efficient for transport and storage purposes is presented. For a deeper discussion and further specifications refer to the H.264 International Standard (9).

2.1.1 High-level structure of an H.264/AVC bit-stream

At a high level structure, a H.264 byte-stream can be represented as sequence of *Byte-streams NAL units* [Fig 10]. The *Network abstraction layer (NAL)* is the last step of the coding process of a single frame. The function of the *NAL* is to format data and provide header information in a way that is appropriate for the transport layers or for storage media, i.e. Format the coded video into *NAL units*

A *NAL unit* is a packet with an integer number of bytes; every *NAL unit* has header bytes, which contain the information about the data it transports. *NAL units* are classified as follow:

- VCL NAL unit: Contains information about encoded pictures from the *video coding Layer (VCL)*

- Non-VCL NAL unit: Contains additional information, like *parameter sets* and *supplemental enhancement information (SEI)*.

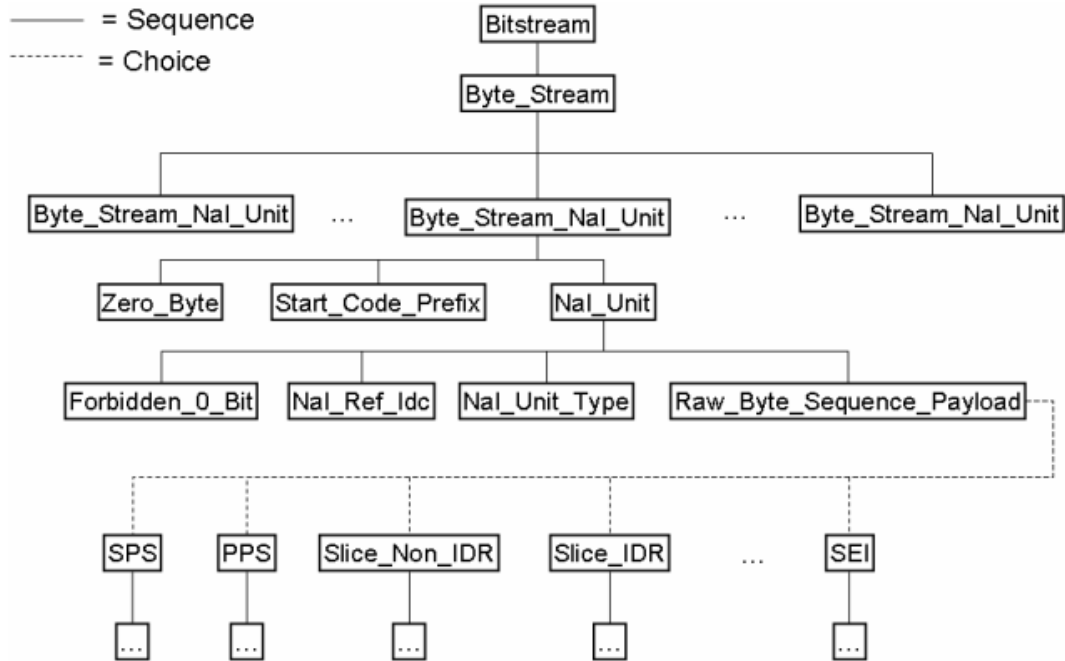


FIG 10 HIGH-LEVEL STRUCTURE OF H.264/AVC BITSTREAM

A set of *NAL units*, that are consecutive in decoding order, have a specific form, known as *access unit*. The decoding of an access unit always results in a decoded picture. Returning to [Fig 10], as mentioned before, a *NAL unit* can contain information regarding a coded picture (or the information from the VCL) and a set of additional information: *Parameter set (SPS, PPS)* and *SEI* messages:

Parameter Sets: Contain information about the coded pictures, like the parameter values. They are divided in:

- *SPS: sequence parameter sets*: Applied to a series of consecutive pictures.
- *PPS: picture parameter sets*: Applied to one individual picture.

This information is transmitted only when necessary; they can also be transmitted on regular intervals to provide robustness to data.

Supplemental Enhancement Information (SEI): Data which contain timing information and other data which enhance the usability of the decoded video. This information should not be necessary for the decoding process.

2.1.2 The video coding layer (VCL)

The video coding layer of H.264 is the responsible of coding the uncompressed digital signal video. The VCL design follows a block-based hybrid video coding approach (a mix of temporal and spatial prediction, as well as the use of transform coding) in which each coded picture is represented in block-shaped units of associated luma and chroma samples called macroblocks. A sequence of macroblocks creates a slice. A picture may be divided into different slices. In H.264/AVC a picture can be seen as a collection of slices which can have different patterns [Fig 11].

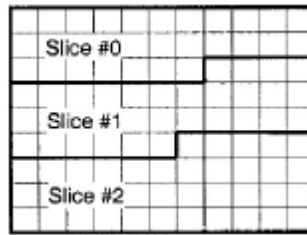


FIG 11 A PICTURE PARTITIONED IN SLICES

In H.264/AVC each slice could be coded with one of five different slices-coding approaches. The three most common approaches are:

- *I slice*: macroblocks are coded with intra prediction, no information needed from other frames.
- *P slice*: macroblocks can be coded using interprediction from prior temporary frames).
- *B slice*: macroblocks can be coded using interprediction from prior and subsequent temporary frames.

[Fig 12] shows a typical coding structure of H.264/AVC, here the arrows indicate prediction dependencies.

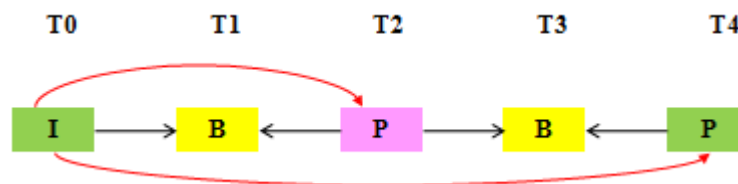


FIG 12 INTERPREDICTION BETWEEN I FRAMES , P FRAMES, AND B FRAMES

2.2 Standard H.264/MVC

One common characteristic of 3-D/Multi-View videos is that they use multiple camera views of the same scene, often referred to as multi-view video (MVV), which is implemented by simultaneously capturing the video streams of several cameras. Since this approach creates large amounts of data to be stored or transmitted to the user, efficient compression techniques are essential for realizing such applications. The straight-forward solution for this would be to encode all the video signals independently using a state-of-the-art video codec such as H.264/AVC. However, a very important feature of these "multi-view" video sequences is that they contain a very large amount of statistical interdependencies between the different views, as each camera captures the same scene from different points. Therefore, combining the time prediction with inter-view prediction more efficient predictions can be reached (10), i.e. higher compression. [Fig 14] show a typical MVC inter-view prediction structure for 5 cameras where the video signals are numbered from S0 to S4 and the separate values of T represent different time instants. The arrows represent the decoding dependencies between the views. For example, view S0 is not dependent on any other view while the view S1 is dependent on views S0 and S2. View S0 is also called the base view and is H.264/AVC compliant.

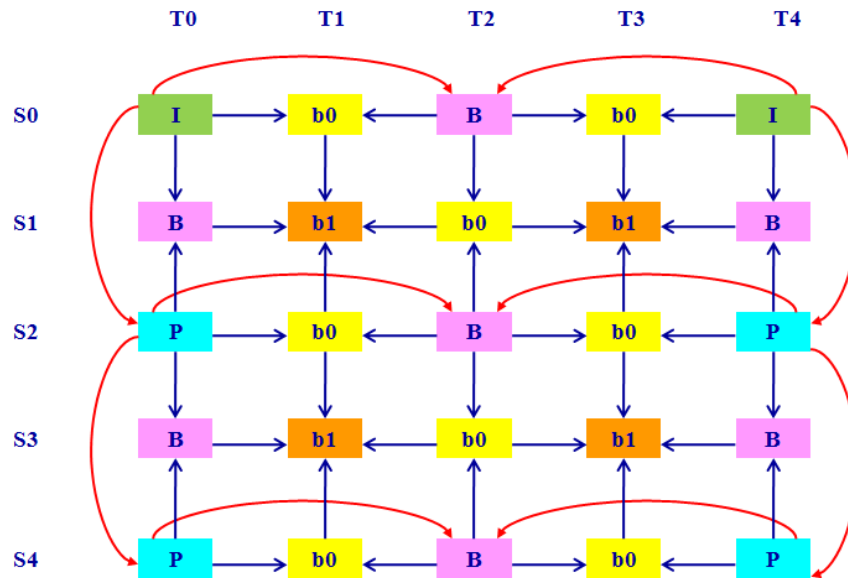


FIG 14 TYPICAL MULTI-VIEW CODING STRUCTURE FOR 5 VIEWS

The Multi-view Video Coding (MVC) extension of the H.264/MPEG-4 AVC video coding standard is described in the Appendix H of (9). MVC enables

efficient encoding of sequences taken simultaneously from multiple cameras using a single video stream. MVC is intended both for encoding stereoscopic video (two views), as well as free viewpoint television and multi-view 3D television (11).

2.2.1 MVC Stream data chain

Basically the multi-view encoder receives N temporally synchronized video streams and generates one bit-stream. The multi-view decoder receives the bit-stream, decodes and outputs the N video scenes. This single data chain stream that contains all or some of the views of a Multi-view video is one of the many attractive features that make H.264/MVC a good coding technology candidate for 3-D/Multi-View video delivery. A possible MVC stream data chain structure is shown in (12) [Fig 15].

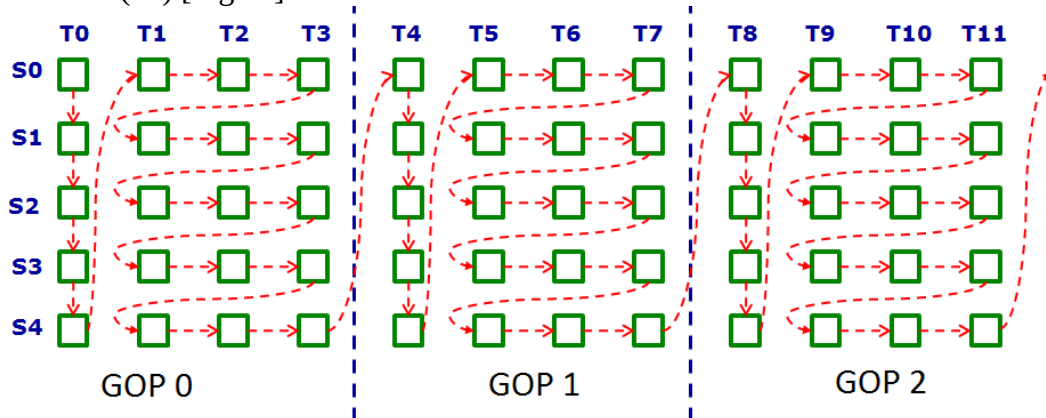


FIG 15 MVC STREAM DATA CHAIN

Pictures of each view are grouped into Group of Pictures (GOP) (e.g. pictures from T4 to T7 in [Fig 15]). To allow synchronization each GOP is started by an I-frame (anchor frame). The anchor frames are used to provide random access as no frame that temporally follows the anchor frame has any reference with the frames temporally preceding the anchor frame as shown in [Fig 14].

2.2.2 H.264/MVC coding structure is adaptive friendly

It is possible to think of cases when it is not necessary to transmit or storage the complete bit-stream, an example being the case when the client needs only the base view or S2 view [Fig 14]. In such cases it will be helpful to perform an

adaptation of the H.264/MVC bit-stream and for that purpose it is necessary to define the concept of *operation point*.

An operation point of an MVC bit-stream represents a certain level of temporal and view scalability. An operation point contains only those NAL units required for a valid bit-stream to represent a certain subset of views at a certain temporal level. In other words, it is possible to save or deliver a view-point by just delivering the NALU of the view-points that contribute on the decoding process of the target view-point, and forgetting those who do not contribute. An operation point may be dependent on other operation points. The information about the characteristics of the operation points and their dependencies is present in the view scalability SEI Message.

To understand this concept, [Fig 16] shows the possible operation points for the coding structure presented in [Fig 14]. Operation point 0 (Op0) allows the decoding of the view-point S0; operation point 1 contains the minimum required view-points to decode view-point S2, but either view-point S2 and S0 could be decoded; operation point 2 contains the minimum required view-points to decode view-point S1, but either view-point S0, S1, S2 can be decoded, and so on.

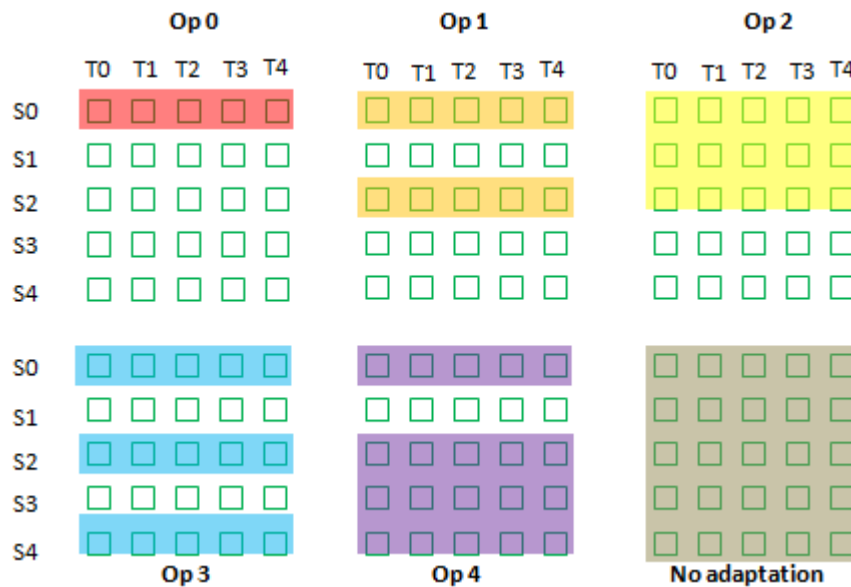


FIG 16 OPERATIONS POINTS FOR THE CODING STRUCTURE IN FIG 14

This useful property opens the doors to many adaptivity processes which introduce enormous advantages in the streaming framework for Multi-View video and they are presented in the next chapters.

3

Media Streaming Technologies and Standards

Streaming media technologies enables the real time or on demand distribution of audio, video and multimedia on the internet. Media streaming is the simultaneous transfer of digital media (video, sound and data) so that it is received as a continuous real-time stream or flow (13). Streamed media is transmitted by a server application and received and displayed in real-time by client applications. These applications can start displaying video or playing audio as soon as enough data has been received and stored in the receiving station's buffer.

This chapter focuses on technical aspects and common definitions of the streaming technologies, needed later to introduce the adaptive HTTP streaming framework for H.264/MVC coded media.

3.1 Live streaming and Video on Demand

Generally, there are two typical scenarios in which streaming technologies are used, one is the streaming of live events (Live streaming) which is supported on broadcast or multicast connections and the other is Video on Demand (VoD) carried by Unicast connections.

3.1.1 Video on Demand and Unicast connections

VoD systems allow the user to choose and watch content stored at the server on demand. The streaming session is interactive and the user can exercise VCR-like control operations (trick play functionality). The “trick play functionality” is provided by signaling between the client and the server, therefore, two-way communication between the server and the client of the VoD system should be available (session), and that means a Unicast connection. Unicast is a one-to-one connection between the client and the server. Unicast uses IP delivery methods, such as Transmission Control Protocol (TCP) and User Datagram Protocol (UDP), which are session-based protocols in which every client has a direct relationship to the server. Each Unicast client that connects to the server takes up additional bandwidth (13). For example, if there are 10 clients all playing 100-kilobits per second (Kbps) streams, those clients as a group are taking up 1,000 Kbps. If there is only one client playing the 100 Kbps stream, only 100 Kbps is are being used. Unicast-based media servers open and provide a stream for each unique user. A Unicast connection scheme is shown in [Fig 17].

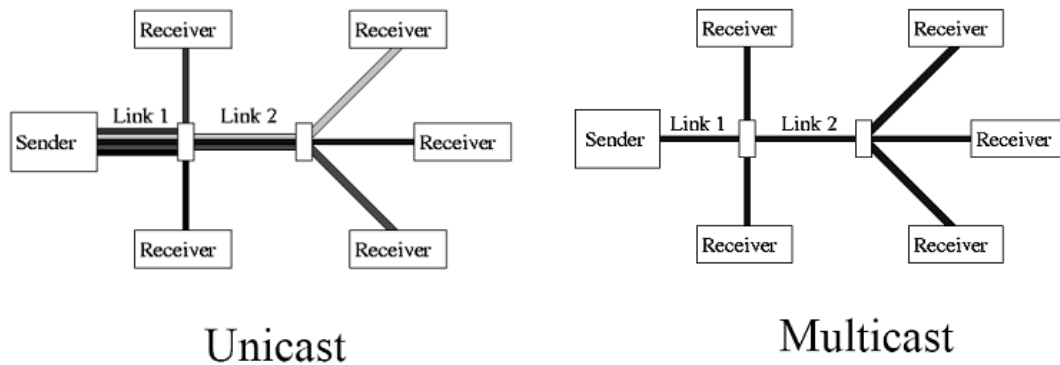


FIG 17 UNICAST VS. MULTICAST CONNECTION IN TERMS OF BANDWIDTH USE (14)

3.1.2 *Live streaming and Multicast connections*

Live streaming should use Multicast Connection for minimize server load and maximize bandwidth performance. The multicast source relies on multicast-enabled routers to forward the packets to all client subnets that have clients listening. There is no direct relationship between the clients and Media server. This is similar to tuning into a station on a radio. Each client that listens to the multicast adds no additional overhead on to the server. In fact, the server sends out only one stream per multicast station. The same load is experienced on the server whether only one client or 1,000 clients are listening.

Multicast on the Internet is generally not practical because only small sections of the Internet are multicast-enabled. Multicast incorporates environments where all routers are multicast-enabled and can save quite a bit of bandwidth (14). It has to be mentioned that the trick play functionality cannot be provided using multicast. A Multicast connection scheme is shown in [Fig 17].

3.1.3 The hybrid solution

A hybrid solution could be the use of web caches for VoD scenarios. A web cache stores copies of documents passing through it; subsequent requests may be satisfied from the cache if certain conditions are met. A cache server both speeds up access to data and reduces demand on an enterprise's bandwidth. To the user, cache servers are invisible; all Internet requests and returned responses appear to be coming from the addressed place on the Internet. Even if there is a huge deployment of cache servers, they are designed particularly for HTTP transported data (media delivery over protocols different than HTTP will require special proxies or caches).

3.2 Components of a streaming framework

Any streaming framework is build over a combination of these minimum components: media coding format, streaming server, transport protocols, buffering schemes, client application layer, and the delivery networks used in the transmission [Fig 18]. Other important components could be added to a streaming framework like Control mechanisms for traffic congestion or QoS, adaptivity services, etc.

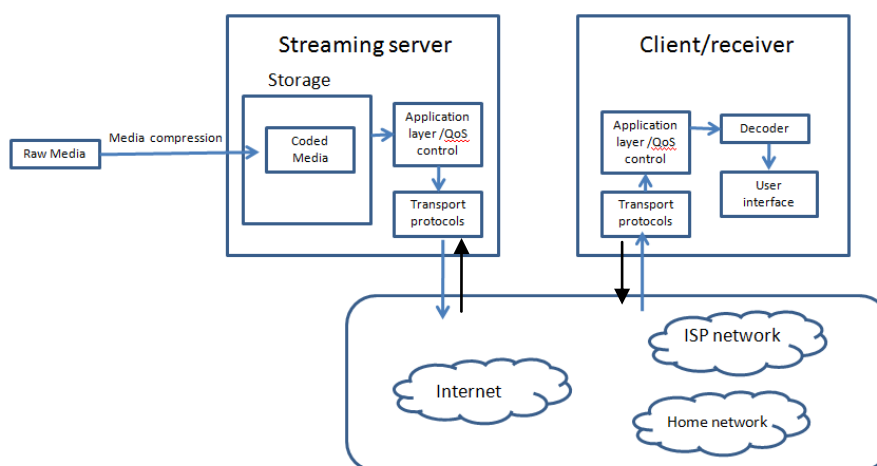


FIG 18 ARCHITECTURE OF A MULTIMEDIA STREAMING SYSTEM

3.2.1 Media coding format

As mentioned in chapter 2, the raw video shot from cameras is compressed using video coding techniques such as H.264/AVC and then stored or immediately delivered. Usually, the coded media is formatted inside a container file. Container files are used to identify and interleave different data types (audio, video, metadata). One or more coding formats or standards should be considered for the media intended to be delivered on a streaming framework. Depending on the coding technology chosen, different services could be offered. For example depending on the configuration parameters set for H.264/AVC coding, different qualities, resolutions can be offered. If the coding standard permits scalability, scalable streaming could be offered, or as mentioned in chapter 2, a H.264/MVC bit-stream could be adapted, therefore, adaptive streaming could be offered.

3.2.2 Streaming server

Streaming servers could be complex servers that have to process multimedia data under timing constraints and offer trick play functionalities, could be simple web servers as in the case of progressive HTTP download or could be a small application or extensions to a standard HTTP/web server (approach used in this thesis work).

The streaming server is in charge of handle media and interaction requests made by users, process the requests, check and apply control mechanism (QoS, Congestion control, etc), and deliver the compressed media or data using a transport protocol.

3.2.3 Control mechanisms

Control mechanisms are a very important part of scalable or adaptive streaming frameworks. It is possible to have control mechanisms at the server side as well as the client side. These mechanisms provide:

- Congestion control by running rate control algorithms. Rate control algorithms can be run at server or client side. These algorithms could

trigger adaptivity or scalability process. An example is the case when server analyzes the network condition and chose a suitable quality or number of services to be transmitted to the client. In the same way, the client can request the server for services or qualities that suits it.

- Error control by using mechanism such as Forward Error Correction (FEC), packet retransmission, error resilient compression and error concealment.

Other way to trigger control mechanisms is measuring Quality of Experience QoE metrics (15), measured at client side. The information can be transmitted to the streaming server to take the pertinent actions. Some of the commonest QoE metrics are:

- Corruption duration metric: Gives information on burst errors etc.
- Rebuffering duration metric: Gives information about “stalls” in playback time due to any involuntary event at the client side.
- Initial buffering duration metric: Gives information about the time form receiving the first media packet until playing start.
- Successive loss of packets: Indicates the number of packet lost in succession per media channel.
- Frame rate deviation: Indicates the playback frame rate information. Frame rate deviation happens when the actual playback frame rate during measurement period is deviated from a pre-defined value.
- Jitter duration: Jitter happens when the absolute difference between the actual playback time and the expected playback time is larger than a predefined value, which is 100 milliseconds.
- Content switching time: Gives information about the time the user attends when request another content.

In the case of guaranteed delivery transports, such as HTTP as used in progressive download or HTTP-based streaming, metrics relating to loss or corruption (such as "Corruption duration", "Successive loss of packets" and "Jitter duration") are not relevant because no corruption can take place as HTTP goes over TCP, which provides reliable, ordered delivery of a stream of bytes.

3.2.4 Buffering schemes

A data buffer, as defined in (16), is a region in memory used to temporally hold data while it is being moved from one place to another. Buffers are typically used when there is a difference between the rate at which data is received and the rate at which it can be processed, or in the case these rates are variable. In a streaming scenario, the delivery of data from the server may be subject to delays known as lag, phenomenon caused when data are scarce (due to interruptions in the connection, overloaded bandwidth, etc). Therefore, media players preload some of the media before start playback (buffering media): the idea is to have a data backup in order to prevent playback stalls or stops. Buffers are usually used in a FIFO (first in, first out) fashioned way, outputting data in the order it arrived.

There are different buffering schemes that can be implemented in a streaming system. They are an important part for the performance and perceived quality of a streaming system; many commercial solutions keep secret their buffering schemes. For example, different buffer schemes could react different to bandwidth variations, triggering proper control mechanisms; they could also manage efficiently the media request in order to efficiently use the random access memory and network resources. (17)

Critical to the performance of streaming systems over best-effort networks, such as the Internet, buffering provides a number of important advantages (18):

- Jitter reduction: Jitter can cause jerkiness in playback due to the failure of same samples to meet their presentation deadlines, and have to be therefore skipped or delayed. The use of buffering effectively extends the presentation deadlines for all media samples, and in most cases, practically eliminates playback jerkiness due to delay jitter.
- Error recovery through retransmissions: The extended presentation deadlines for the media samples allow retransmission to take place when packets are lost.
- Error resilience through Interleaving: Losses in some media streams, especially audio, can often be better *concealed* if the losses are isolated instead of concentrated. The extended presentation deadlines with the use of buffering allow interleaving to transform possible burst loss in the channel into isolated losses, thereby enhancing the concealment of the subsequent losses.

- Smoothing throughput fluctuation: Since time varying channel gives rise to time varying throughput, the buffer can provide needed data to sustain streaming when throughput is low. This is especially important when streaming is performed using TCP (or HTTP), since the server typically does not react to a drop in channel throughput by reducing media rate.

The benefits of buffering do come at a price though. Besides additional storage requirements at the streaming client, buffering also introduces additional delay before playback can begin or resume (after a pause due to buffer depletion).

3.2.5 Streaming Client application

The streaming client can be implemented as an external application which feed a media player with coded media that has been properly processed and taken from the transport layer. It also handles signaling and interactions with the streaming server. On the other hand, the streaming client can be a module implemented inside a media player as shown in [Fig 19]. The streaming module will be in charge of feeding the decoders with the media arriving at the transport layer; it also handles the interactions and signaling between clients and streaming server. Different buffering schemes can be implemented at the streaming client.

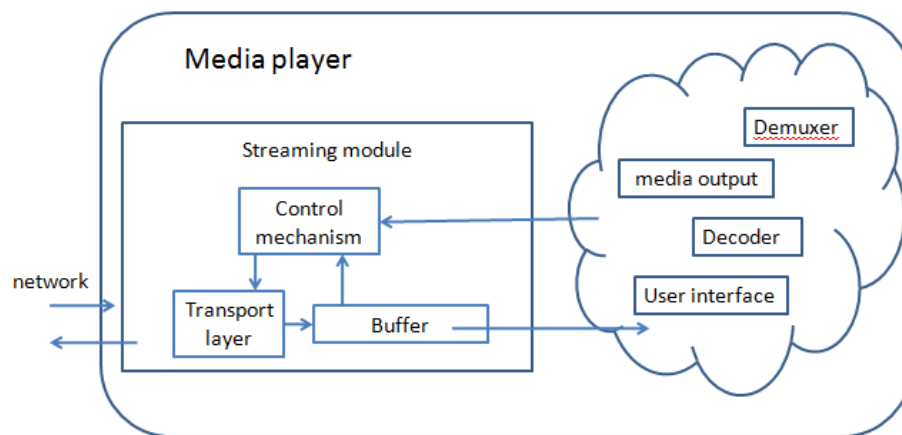


FIG 19 STREAMING CLIENT CAN BE A MODULE IMPLEMENTED INSIDE A MEDIA PLAYER

3.3 Protocols for media streaming

This section briefly describes the network protocols for streaming media through Internet. First, we review the major Internet protocols IP, TCP and UDP followed by traditional media delivery and control protocols (RTP, RTSP, RTCP). And finally HTTP as a streaming protocol will be introduced.

3.3.1 *Internet protocol IP*

The Internet Protocol (IP) is the principal communications protocol used for relaying datagrams (packets) across an internetwork using the Internet Protocol Suite. Responsible for routing packets across network boundaries, it is the primary protocol that establishes the Internet. The Internet Protocol only provides best effort delivery and its service can also be characterized as unreliable. In network architectural language it is a connection-less protocol. The lack of reliability allows any of the following fault events to occur: data corruption, lost data packets, duplicate arrival and out-of-order packet delivery. On top of IP are the end-to-end transport protocols, where Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) are the most important.

3.3.2 *TCP and UDP protocols*

TCP provides reliable byte-stream services. It guarantees delivery via retransmissions and acknowledgements. On the other hand, UDP is simply a user interface to IP, and is therefore unreliable and connectionless, and it has no mechanism within the protocol to guarantee delivery. It is up to the receiving application to detect loss or corruption and recover data using error correction techniques. If data is lost, the stream may suffer a dropout.

Reliable protocols, such as TCP, guarantee correct delivery of each bit in the media stream, and no corruption can take place. However, they accomplish this with a system of timeouts and retries, which makes them more complex to implement. It also means that when there is data loss on the network, the media stream stalls while the protocol handlers detect the loss and retransmit the missing data. Clients can minimize this effect by buffering data for display. While delay due to buffering is acceptable in video on demand scenarios, users of live

scenarios and videoconferencing will experience a loss of fidelity if the delay that buffering contributes to exceeds 200 ms (19). TCP is ideal for High Definition streaming because, as mentioned, no corruption can take place. Nevertheless, TCP is a rate controlled protocol, feature than can introduce additional delay and unexpected variance in the stream bit-rate; this problem will be treated later in this chapter. Anyway, if we take a look at the current grow of internet, the network share of video on it, the network diversity and the users expectation on QoS (refer to chapter 1) it is not foolish to think TCP as a good protocol for video streaming if proper adaptivity mechanism are implemented over it, as will be shown in chapter 4 and the following sections.

3.3.3 RTP/RTSP/SDP protocols

RTSP (Real-Time Streaming Protocol) defined in RFC 2326 (20) is an application layer protocol for control over the delivery of data with real-time properties. This protocol is intended to control multiple data delivery sessions, provide a means for choosing delivery channels such as UDP, multicast UDP and TCP, and provide a means for choosing delivery mechanisms based upon RTP (14). RTSP is a good example of a traditional streaming protocol. RTSP is defined as a stateful protocol, which means that from the first time a client connects to the streaming server, and until the time it disconnects from the streaming server, the server keeps track of the client's state. The client communicates its state to the server by issuing it commands such as PLAY, PAUSE or TEARDOWN. After a session between the client and the server has been established, the server begins sending the media as a steady stream of small packets (with the delivery mechanism based upon RTP). The size of a typical RTP packet is 1452 bytes. In RTSP the packets can be transmitted over either UDP or TCP transports (the latter is preferred when firewalls or proxies block UDP packets, but can also lead to increased latency) (6). These protocols do not enable real-time services, only the underlying network can do this. However, they provide functionalities that support real-time services. RTP does not guarantee QoS or reliable delivery, but provides support for applications with time constraints by providing a standardized framework for common functionalities such as time stamps, sequence numbering, and payload specification. RTP enables detection of lost packets. The protocol stack for RTP/RTSP streaming is shown in [Fig 20].

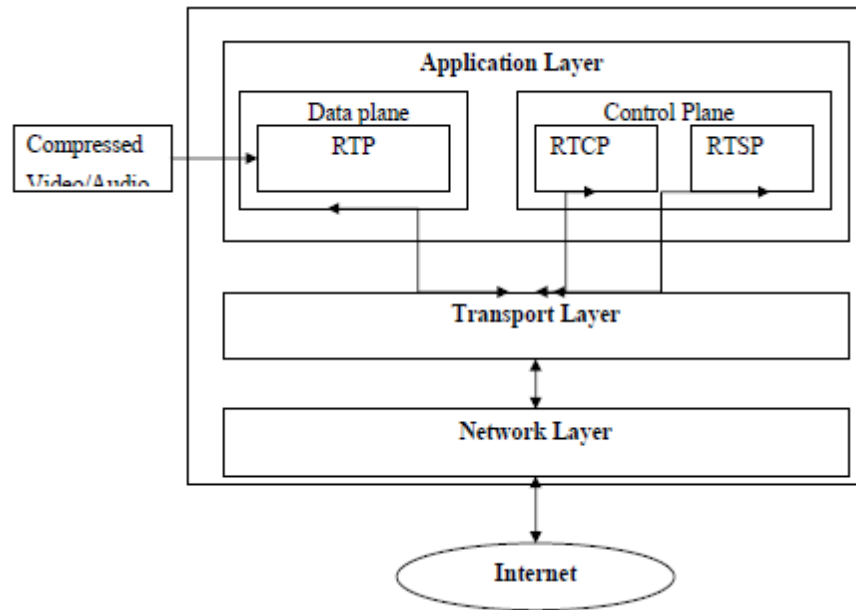


FIG 20 PROTOCOL STACK FOR RTSP/RTP STREAMING

Finally, the Session Description Protocol (SDP) is a format for describing streaming media initialization parameters. The IETF published a revised specification, IETF Proposed Standard as RFC 4566 (21) in July 2006. The Session Description Protocol (SDP) provides information describing a session, for example whether it is video or audio, the specific codec, bit rate, duration, etc. (22)

3.3.4 HTTP/1.1 protocol

The Hypertext Transfer Protocol (HTTP) (23) is a networking protocol for distributed, collaborative, hypermedia information systems. HTTP is the foundation of data communication for the World Wide Web. The standards development of HTTP has been coordinated by the Internet Engineering Task Force (IETF) and the World Wide Web consortium.

HTTP functions as a request-response protocol in the client-server computing model. In HTTP, a web browser, for example, acts as a *client*, while an application running on a computer hosting a web site functions as a *server*. The client submits an HTTP *request* message to the server. The server, which stores content, or provides resources, such as HTML files and images, or generates such

content as required, or performs other functions on behalf of the client, returns a response message to the client. The response contains completion status information about the request and may contain any content requested by the client in its message body. The HTTP often benefit from web cache servers that deliver content on behalf of the original, so-called origin server to improve response time. HTTP is a stateless protocol. A stateless protocol does not require the server to retain information or status about each user for the duration of multiple requests.

HTTP resources are identified and located on the network by Uniform Resource Identifiers (URIs) or, more specifically, Uniform Resource Locators (URLs) using the http or https URI schemes.

HTTP is an Application Layer protocol designed within the framework of the Internet Protocol Suite. The protocol definitions presume a reliable Transport Layer protocol for host-to-host data transfer. The Transmission Control Protocol (TCP) is the dominant protocol in use for this purpose.

In HTTP/1.0 and since, the first line of the HTTP response is called the status line and includes a numeric status code (such as "404") and a textual reason phrase (such as "Not Found"). The way the user agent handles the response primarily depends on the code and secondarily on the response headers. Custom status codes can be used since, if the user agent encounters a code it does not recognize, it can use the first digit of the code to determine the general class of the response.

In HTTP/0.9 and 1.0, the connection is closed after a single request/response pair. In HTTP/1.1 a keep-alive-mechanism was introduced, where a connection could be reused for more than one request. Such persistent connections reduce lag perceptibly, because the client does not need to re-negotiate the TCP connection after the first request has been sent. Version 1.1 of the protocol made bandwidth optimization improvements to HTTP/1.0. For example, HTTP/1.1 introduced chunked transfer encoding to allow content on persistent connections to be streamed, rather than buffered. HTTP pipelining further reduces lag time, allowing clients to send multiple requests before a previous response has been received to the first one. Another improvement to the protocol was byte serving range, which is when a server transmits just the portion of a resource explicitly requested by a client.

In HTTP, the header fields contain the operating parameters of an HTTP request or response. With the request or response line (first line of the message), they form the message header. The header fields can define various characteristics of the

data transfer. Header fields start with the field name, followed by the field value. A List of HTTP header fields can be found in (23)

3.4 HTTP streaming

This section introduce a Brief introduction to HTTP as a media delivery protocol, it explains several strong reasons why industry in recent years is shifting from the classical streaming protocols (RTSP, MMS, RTMP, etc.) into HTTP as a protocol for media streaming. Nevertheless, it is important to differentiate plain HTTP download or “progressive download” from true HTTP streaming or “HTTP adaptive bitrate streaming”.

3.4.1 HTTP progressive download

Another common form of media delivery on the Web today is progressive download, which is nothing more than a simple file download from an HTTP Web server. Progressive download is supported by most media players and platforms. The term "progressive" stems from the fact that most player clients allow the media file to be played back while the download is still in progress, before the entire file has been fully written to disk. Clients that support the HTTP 1.1 specification can also seek to positions in the media file that have not been downloaded yet by performing byte range requests to the Web server (assuming that it also supports HTTP 1.1) (6). Popular video sharing Web sites on the Web today, including YouTube, Vimeo, MySpace, and MSN Soapbox, almost exclusively, use progressive download.

Unlike streaming servers that rarely send more than 10 seconds of media data to the client at a time, HTTP Web servers keep the data flowing until the download is complete. For example, if progressively downloaded video is paused at the beginning of playback, the entire video will eventually be downloaded to the browser cache, allowing to smoothly play the whole video. There is a downside to this behavior as well. If at 30 seconds of seeing a fully downloaded 10 minute video, client does not like it and quit the video, both client and content provider have just wasted 9 minutes and 30 seconds worth of bandwidth. The other downside is it makes extremely easy for someone to steal the media content (23).

Delivering a progressive download via HTTP is as simple as creating a media file, uploading the file to a web server, creating an HTML reference file with an HTTP link to the file, and publishing the link. The file will begin playing as it is downloaded (cached to the client machine) in either a stand-alone player or an embedded player.

3.4.2 HTTP adaptive bitrate streaming

Another approach that seems to incorporate both the advantages of using a standard web protocol (section 1.4) and the ability to be used for streaming, even live content, is the HTTP adaptive bitrate streaming. HTTP adaptive bitrate streaming is based on HTTP progressive download, but contrary to the previous approach, here the files are very small. In a typical adaptive streaming implementation, the video/audio source is cut into many short “segments” or “chunks” and encoded to the desired delivery format. Chunks are typically 2-to-10 seconds long. At the video codec level, this typically means that each segment contains a certain number of GOP (Group of Pictures and has no dependencies on past or future chunks/segments. This allows each segment to later be decoded independently of other segments.

The encoded segments are hosted on a HTTP Web server. A client requests the segments from the Web server in a linear fashion and downloads them using plain HTTP progressive download. As the segments are downloaded to the client, the client plays back the sequence of segments in linear order. Because the segments are carefully encoded without any gaps or overlaps between them, the segments play back as a seamless video.

The "adaptive" part of the solution comes into play when the video/audio source is encoded at multiple bit rates or in the case of H.264/MVC multiple operation points (section 2.2.2 and figure Fig 16), generating multiple segments of various sizes for each 2-to-10-seconds of video. The client can now choose between segments of different sizes. Because Web servers usually deliver data as fast as network bandwidth allows them to, the client can easily estimate user bandwidth and decide to download larger or smaller segments ahead of time [Fig 21]. The size of the playback/download buffer is fully customizable.

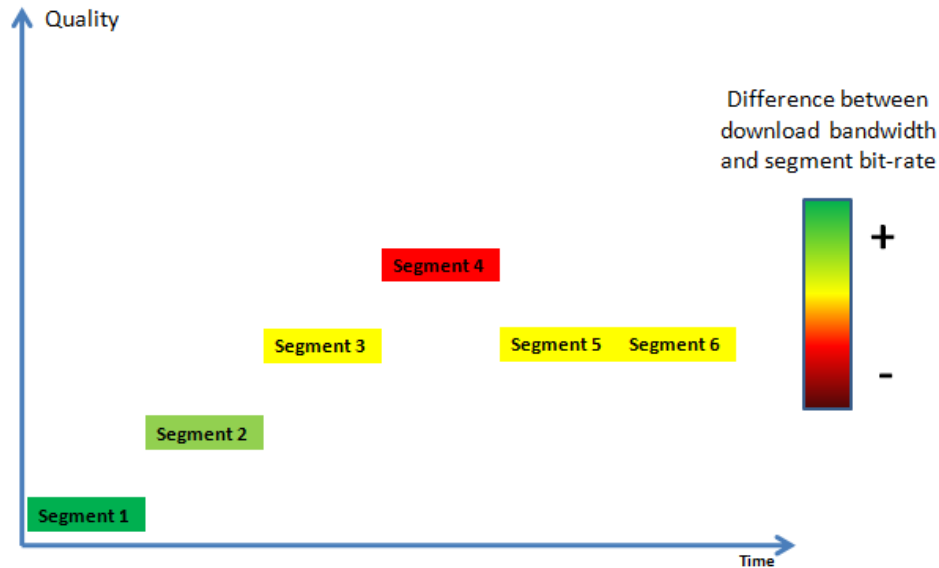


FIG 21 HTTP ADAPTIVE STREAMING

Adaptive streaming, like other forms of HTTP delivery, offers the following (and more) advantages over traditional streaming to the content distributor: (6)

- It is cheaper to deploy because adaptive streaming can use generic HTTP caches/proxies and does not require specialized servers at each node.
- It offers better scalability and reach, reducing "last mile" issues because it can dynamically adapt to inferior network conditions as it gets closer to the user's home.
- It lets the audience adapt to the content, rather than requiring content providers to guess which bit rates are most likely to be accessible to their audience.

It also offers the following benefits for the user and more:

- Fast start-up and seek times because start-up/seeking can be initiated on the lowest bit rate before moving to a higher bit rate.
- No buffering, no disconnects, no playback stutter (as long as the user meets the minimum bit rate requirement).
- Seamless bit rate switching based on network conditions and CPU capabilities.

3.4.3 HTTP streaming. SoA in commercial solutions

In recent years, the internet has become an important channel for delivery of multimedia. As the HTTP protocol is widely used on the internet, it has recently been used extensively for the delivery of multimedia content. 2010 became a year of tough struggle between proprietary solutions. [TABLE 1] reviews briefly the features of the HTTP adaptive streaming technologies of the major contenders in the media delivery business (24).

features	Microsoft Smooth Streaming	Adobe HTTP Dynamic Streaming	Apple HTTP Live Streaming
Trick mode	yes	yes	yes
Transfer protocol	http	http	http
Supported client platform	Silverlight 2, Xbox 360, Moonlight for Linux	Flash Player 10.1 / Adobe AIR 2	iOS (3.0 or later), Quicktime X
default chunk/fragment size	1-4 sec	4 sec	10 sec
Error detection / correction	via tcp/ip, http;	via tcp/ip, http;	via tcp/ip, http;
Client manifest	.ismc - xml format	.f4m - xml format	m3u8 (extension of m3u)
Stream control	by client	by client	by client
Caching, proxy possible	yes	yes	yes
Disk format can be used by other players?	no	no	no
Video codec	H.264 / VC-1	H.264 / VP6	H.264
Physical files on server	one file per segment	one file per segment	each chunk of each segment in extra file
Cost per Streaming Server	Included in Windows Server. Other vendors can also implement.	5,363,60 EUR (Adobe Flash Media Interactive Server)	(Mac OS X Server: 499,00 €)
Stateless Server Connection	yes, based on persistent HTTP	yes, based on persistent HTTP	yes, based on persistent HTTP
Maximum Bit rate	Unlimited. Proven to support 1080p video on Xbox Live movie streaming.	Unlimited	unlimited
Client Programming Platform	Any. Native porting kits available (ANSI C). Silverlight supports .NET languages such as C#.	Flash - ActionScript	Any
Bandwidth estimation, rules adopted in order to perform switching between chunks/levels	Client implementation choice. This is a point of product differentiation.	choice of implementation, general functions provided	tbd
Available scalability levels: @quality, @spatial, @temporal, mix	Content provider and encoder choice. PIFF and SSTP do not constrain these choices.	Content provider and encoder choice	tbd

TABLE 1 OVERVIEW OF MAJOR HTTP ADAPTIVE STREAMING SYSTEMS

Interestingly, all major commercial and standard solutions for HTTP adaptive streaming use a sort of manifest file to introduce the media resource to the client. The manifest file is a structured collection of metadata and segments URL location. The HTTP-Streaming client requests and downloads the manifest file prior the start of the streaming session. Manifest file is available in different formats. E.g. 3GPP's adaptive HTTP streaming (AHS) defines a Media Presentation Description (MPD), Adobe's Dynamic HTTP Streaming, uses the proprietary FMF manifest, Apple HTTP live streaming uses a M3U playlist file, Microsoft's Smooth streaming also use a server manifest file (i.e. SMIL document). The manifest file introduced in this framework follows the guidelines of Adaptive HTTP streaming. 3GPP's specification (16).

3.5 HTTP streaming. 3GPP's Standard

3GPP has proposed a guideline for an Adaptive HTTP-Streaming protocol aimed to provide a streaming service (15). This enables delivering content from standard HTTP servers to an HTTP-Streaming client and enables caching content by standard HTTP caches. [Fig 22] shows the architecture for Adaptive HTTP streaming. 3GPP's specification only deals with the specification of the interface 1 [Fig 22] between the HTTP-Streaming Client and the HTTP-Streaming Server. All other interfaces are out-of-scope of this specification.

3GPP propose that the HTTP-Streaming Client has access to a Media Presentation Description (MPD) file (a sort of *manifest file*). An MPD provides sufficient information for the HTTP-Streaming Client to provide a streaming service to the user by sequentially downloading media data from an HTTP server and rendering the included media appropriately.

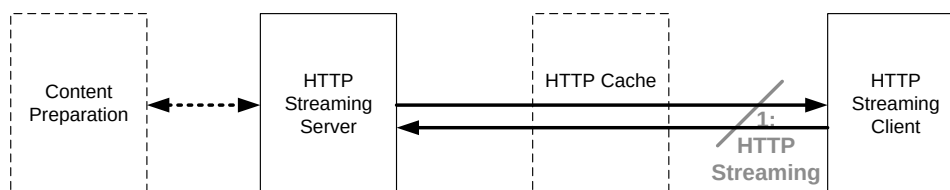


FIG 22 SYSTEM ARCHITECTURE FOR ADAPTIVE HTTP STREAMING

To initiate the streaming service to the user, the HTTP Streaming Client establishes a media presentation by downloading the relevant metadata and subsequently the media data. The MPD is defined in section 3.6.

3.6 The Media presentation description file (MPD)

A media presentation is a structured collection of data that is accessible to the HTTP-Streaming Client. The HTTP-Streaming client requests and downloads media data information to present the streaming service to the user. A media presentation consists of:

- A sequence of Periods.
- Each Period contains one or more Representations from the same media content.

- Each Representation consists of one or more segments.
- Segments contain media data and/or metadata to decode and present the included media data.

[Fig 23] shows graphically a high level structure of a MPD file.

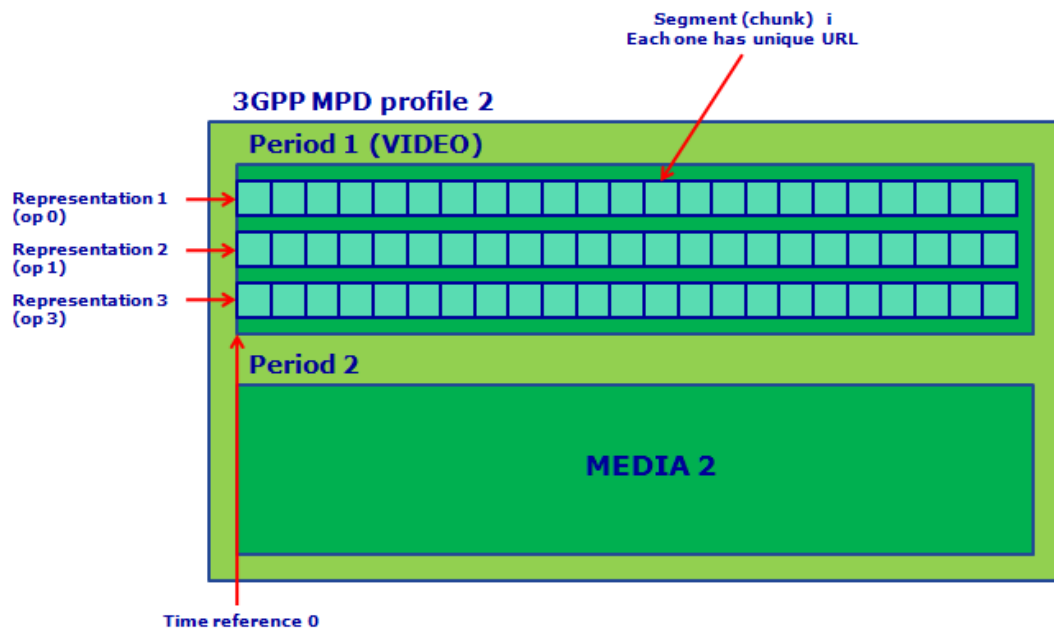


FIG 23 MPD STRUCTURE PROPOSED BY 3GPP

The media presentation may be OnDemand type or Live type. The MPD attribute type provides the type of the media presentation. If the type of the media presentation is OnDemand then the media presentation start time is 0. If the type of the media presentation is Live then the media presentation start time is equal to the MPD attribute availabilityStart.

The Media Presentation Description (MDP) contains metadata required by the client to construct appropriate URIs to access segments and to provide the streaming service to the user. The media presentation may be available in different representations (different bitrates, languages, media components, etc.), and may be on-demand or live. The MPD contains information that enables the client to build the URL to access any provided segment (or parts thereof) of the presentation. The MPD is an XML-document that is formatted according to the XML schema provided in (15).

3.6.1 *Period*

A media presentation consists of one or more periods corresponding to the *Period* elements in the MPD. Each period has an attribute *start*. For OnDemand the *start* attribute of the first period shall be 0. For any other period the *start* attribute specifies the media presentation time of the first sample of each period in the media presentation relative to the *start* time of the first period in this media presentation. Each period extends until the start of the next period, or the end of the media presentation in the case of the last period. Period start and end times are precise. They reflect the actual timing resulting from playing the media in the period.

3.6.2 *Representation*

Each period may define one or several *Representations*. A representation is one of alternative choices differing e.g. by bitrate, resolution, language, etc. A representation starts at the *start* of the Period and continues to the end of the Period. The Representation consists of one or multiple segments. Each Representation either contains an Initialization Segment or each Media Segment in the Representation is self-initializing. Media *components* are time-continuous across boundaries of consecutive Media Segments within one representation. The timing within each representation is relative to the start time of the period that contains this representation.

3.6.3 *Segments*

A Segment is defined as a unit that can be uniquely referenced by an http-URL element in the MPD. A Media Segment contains media components that are either described within this Media Segment or it is described by the Initialization Segment of this Representation. In addition, a Media Segment:

- Has assigned a unique MPD URL Element.
- Has assigned a start time relative to the start of the representation provided by the MPD such that the client can download the segment. The start time

shall be drift-free between the time indicated in the MPD and internal clock of the media data, i.e. the accuracy of the start time documented in the MPD relative to the internal clock does not depend on the position of the segment in the representation.

- Provides random access information, namely if you can randomly access this representation within this segment and if yes, how to randomly access the representation within this segment. There is no requirement that a segment starts with a random access point (RAP). But it is possible to signal in the MPD that all segments within a representation start with a RAP.
- Contains sufficient information on how to accurately present each contained component in the representation without accessing any previous media segment in this representation.
- May provide metainformation for faster accessing subsets of the segment.

3.6.4 Media Presentation Description Updates

The MPD may be changed during the media presentation. If the attribute *updatePeriod* is provided then the client derives the *CheckTime* as the sum of the time of its last requested update of the MPD and the *minimumUpdatePeriod*. If the *updatePeriod* is not provided, external means may be used to deduce the *CheckTime* of the MPD. If the server changes the MPD, then the changes to the MPD must be such that the updated MPD is compatible with the previous MPD in the following sense:

- Clients may immediately begin using the Media Presentation Description without synchronization, since it is compatible with the old one for times before the update time.
- The update time need not be synchronized with the time at which the change to the presentation metadata takes place: i.e. changes to the presentation metadata may be advertised in advance.

4

Description of the Adaptive HTTP Streaming Framework for H.264/MVC Coded Media

With all the motivations and basic definitions of HTTP adaptive streaming systems and H.264/MVC introduced in the precedent chapters, it is possible to introduce an advanced, efficient and flexible adaptive http streaming framework for H.264/MVC coded media. First of all, a brief description of the capacities and properties of the entire framework is presented, then, the following sections will explain how the framework works and how each one of those capacities or properties were implemented, achieved or constructed.

4.1 Framework overview

As discussed in section 3.1, there are two typical scenarios in which streaming technologies are used, one is the streaming of live events and the other is provision of contents on demand (Video-On-Demand, VoD). This framework was conceived for VoD scenarios but could be extended to live scenarios also. Using the MPD file introduced in section 3.6. this framework extends the adaptivity and network scalability concept from client side into client and Content Delivery Network side, opening the doors for new ways of traffic control, congestion control and management of network resources. The media providers could also benefit of QoS control and services offered control.

The standard coding considered was H.264/MVC, H.264/AVC extension that allows us to encode together multiple views of a scene. This choice permitted this framework to extend the adaptivity concept beyond the grounds of quality and resolution, reaching possible adaptation at operation-point level (section 2.2.2) i.e. at multi-view point level, and 2-D to 3-D level.

The framework also permits control mechanisms. These mechanisms are in charge of trigger scalable and adaptive processes. In this framework, it is possible to have control mechanisms at the server side as well as the client side. Both will be explained in this chapter, and moreover implemented in chapter 6.

Buffering schemes (section 3.2.4) are a client implementation choice, important to differentiate client products. This framework is compatible with different buffering schemes. In this thesis, an efficient buffer scheme was implemented. This buffer scheme minimize the starting buffering time and dynamically adapt its size to the segments size, always having at least one segment buffered to avoid

jitter and stalls; it also trigger adaptivity mechanisms. The details of such buffer functions are not provided in this thesis work as they contain sensible information for STM, and innovative solutions proposed are potentially patent pending. Some demos showing the buffering scheme capabilities are shown in chapter 7.

The framework is flexible about the format in which the media files and segments are hosted by the web server. A call for proposals on HTTP streaming of MPEG media has been recently issued; at the time of writing discussion on extensions for media file format and for MPD is in progress. In our implementation we rely on 3GPP specifications and the MVC elementary stream containing all the views is splitted in multiple different files. Each file corresponds to a segment which is filtered at server side to generate substreams according to the MPD representation requested by the client. Segment size is flexible and can go from one GOP to several seconds. As for 3GPP the protocol used to deliver the media segments is HTTP/1.1. by continuously using partial HTTP requests, only the relevant parts of the media segment may be accessed for improved user experience and low start-up delays.

Playback tricks functionality is also part of the features of this framework.

[TABLE 2] provides a framework overview, in general, the framework allow the next adaptivity and scalability capabilities:

- On the fly controllability by the server in order to adapt the stream to the server constraints or user's category.
- On the fly traffic redistribution to enhance QoS and network resources.
- Playback tricks and allow the user to select his preferred 2D/3-D view-points.
- Adaptation at client side: automatically choose the best stream to fit its requirements on energy consumption, resolution and band.

features	Proposed Solution
Live streaming possible	Not yet
VoD possible	yes
Trick mode	yes
Transfer protocol	http
Supported client platform	VLC compatibles
default chunk/fragment size	from GOP size to minutes
Error detection / correction	via TCP/IP,HTTP
Client manifest	3GPP standard (XML)
Stream control	Client and server
Caching, proxy possible	yes
Disk format can be used by other players?	waiting MPEG solution
Video codec	H.264/MVC
Physical files on server	segment in extra file until MPEG standard defined
Cost per Streaming Server	works over major open source HTTP servers
Stateless Server Connection	yes, based on persistent HTTP
Maximum Bit rate	unlimited
Client Programming Platform	Any
Bandwidth estimation, rules adopted in order to perform switching between chunks/levels	Yes, client implementation choice.
Available scalability levels:@quality, @spatial, @temporal, mix	Content provider and encoder choice

TABLE 2 HTTP ADAPTIVE FRAMEWORK OVERVIEW

4.2 The manifest file or Media Presentation Description file

Before describing how the framework works, it is important to explain that this framework as other major commercial solutions uses a manifest file. The manifest file is a structured collection of data that is accessible to the HTTP-Streaming Client. The HTTP-Streaming client requests and downloads the *manifest file* with media data information to present the streaming service to the user. The format for this manifest file is the one proposed by 3GPP in (15) and which structure is explained in section 3.6. From now on it will be called *Media Presentation Description file (MPD)* according to 3GPP standard.

4.2.1 MPD for H.264/MVC contents

One of the most important features of a H.264/MVC stream is its capability to be divided into lighter sub-streams. This property is a solution to the possible case when clients are only interested in a subset of View-Points of all the View-Points contained in the main MVC bit-stream, letting the server storage or transmit the sub-stream instead of the complete bit-stream. From now on these sub-streams will be referred as operation points. This interesting and useful property and the concept of operation point are deeply explained in section 2.2.2.

The MPD for MVC content can exploit the concept of operation points to offer scalable representation of the video. Representations (section 3.6.2) are the different alternative choices differing e.g. by bitrate, resolution, language, and now operation points.

For example, the MPD structure shown in [Fig 23, page 35] tells client that the media requested has two *periods* (e.g. corresponding to two different episodes of a TV show). It also informs client that *Period 1* has 3 different *representations* (e.g. *representation 1* offers a sub-bitstream with only the 2-D base view, *representation 2* offers the entire bitstream that contains View-Points 0 and 1 letting the client experience a 3-D effect, and finally the third representation could be a different bit-stream with only the base-view and lower quality). Each representation can contain information about bit-rate, segments length, etc and the URLs to each segment (section 3.6.3).

[Fig 24] Shows an example of MPD file. This MPD file offers just one representation of a media resource called “Ballroom”, particularly, the operation point 0, as can be seen in the URLs of each segment. The representation is constituted by eleven segments of 0.96 seconds of duration each one and VGA resolution (640 x 480). The MPD files are written in XML language and follow the guideline presented in 3GPP standard (16).

```

- <MPD type="OnDemand" minBufferTime="0.96S">
- <ProgramInformation moreInformationURL="http://www.st.com">
  <Title>Ballroom @ 8 views</Title>
  <Source>STmicroelectronics</Source>
  <Copyright>STmicroelectronics, All rights reserved</Copyright>
</ProgramInformation>
- <Period segmentAlignmentFlag="True">
- <Representation bandwidth="642348" width="640" height="480" mimeType="video/mp4" startWithRAP="False">
- <SegmentInfo duration="0.96S">
  - <!--
    InitialisationSegmentURL sourceURL="FIFA-v96k-a48k_i.3gp"/
  -->
  <Url sourceURL="http://mvc.media.net/ballroom_8views/profile_1/oppoint_0/chunk_000"/>
  <Url sourceURL="http://mvc.media.net/ballroom_8views/profile_1/oppoint_0/chunk_001"/>
  <Url sourceURL="http://mvc.media.net/ballroom_8views/profile_1/oppoint_0/chunk_002"/>
  <Url sourceURL="http://mvc.media.net/ballroom_8views/profile_1/oppoint_0/chunk_003"/>
  <Url sourceURL="http://mvc.media.net/ballroom_8views/profile_1/oppoint_0/chunk_004"/>
  <Url sourceURL="http://mvc.media.net/ballroom_8views/profile_1/oppoint_0/chunk_005"/>
  <Url sourceURL="http://mvc.media.net/ballroom_8views/profile_1/oppoint_0/chunk_006"/>
  <Url sourceURL="http://mvc.media.net/ballroom_8views/profile_1/oppoint_0/chunk_007"/>
  <Url sourceURL="http://mvc.media.net/ballroom_8views/profile_1/oppoint_0/chunk_008"/>
  <Url sourceURL="http://mvc.media.net/ballroom_8views/profile_1/oppoint_0/chunk_009"/>
  <Url sourceURL="http://mvc.media.net/ballroom_8views/profile_1/oppoint_0/chunk_010"/>
  </SegmentInfo>
</Representation>
</Period>
</MPD>

```

FIG 24 MPD XML EXAMPLE WITH ONE REPRESENTATION

[Fig 25] Shows another example of MPD file. This MPD file offers four representations of the media resource called “Ballroom”, particularly, the operation points 0,1,2,3,4. The representations are collapsed in this XML example but they are constituted by eleven segments of 0.96 seconds of duration each one and VGA resolution (640 x 480). This MPD file offers more flexibility to the client than the previous one. The client could choose or switch between representations either due to a personal choice of the user or automatically due to client (adaptivity); e.g. control mechanisms can trigger a switch from a 3-D representation into a 2-D representation if certain bandwidth requisites are not meet.

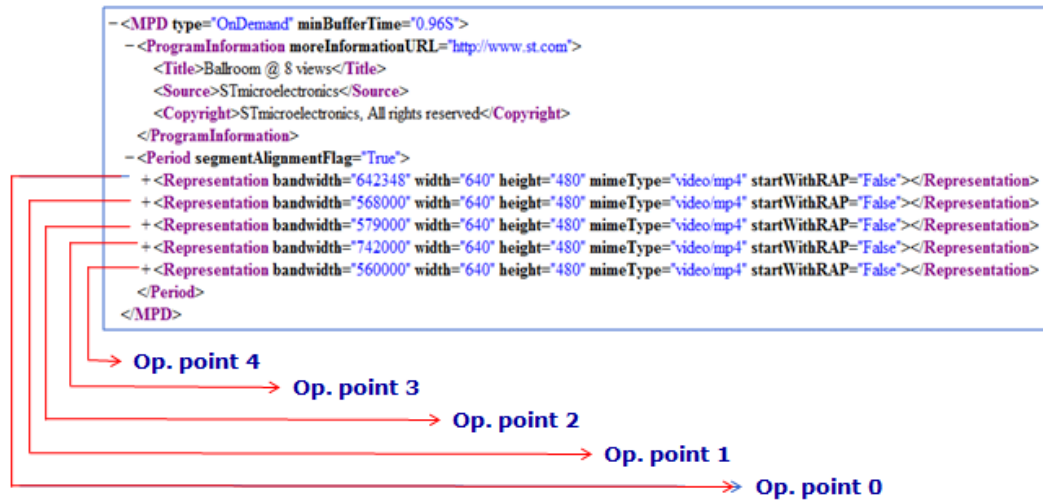


FIG 25 MPD XML EXAMPLE WITH FOUR REPRESENTATIONS

4.2.2 MPD profiles

[Fig 24] and [Fig 25] shows two different MPD files for the same media resource. In general, a media resource can have more than one MPD file, which let the server or content provider to decide the number of services that will be offered or hidden for a certain media resource. This interesting property, combined with the fact that the client's MPD file can be updated inside a streaming session, will open the doors to new server's mechanisms of control and network scalability. The different MPD files for a certain media resource are known as MPD profiles.

For example, [Fig 26] shows a graphic of MPD profiles for the same media resource. The media resource in consideration has 3 View-Points. The first profile allows access only to the sub-stream that contains the base-view (operation point 0). The second profile allow access to two operation points (0,1) and therefore, to View-points 0 and 1, (or the 3-D view composed from 0 and 1). Profile 3 has all the previous operations points plus the original bit-stream (op.3) that offers all the possible views in one bit-stream, (it will require more bandwidth but will offer smother changes between view-points). Finally there is a fourth profile offering additional services like full-HD resolution, etc.

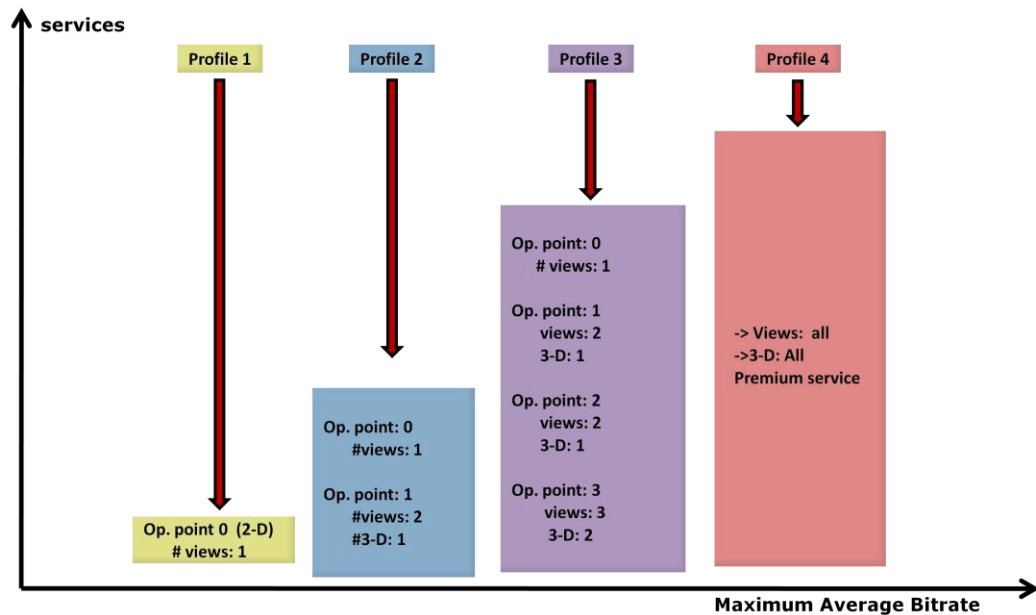


FIG 26 DIFFERENT MPD PROFILES FOR THE SAME MEDIA RESOURCE

4.3 Framework architecture

As shown in [Fig 27] the proposed framework is primarily constituted by an adaptive HTTP streaming enabled server and any number of adaptive HTTP streaming enabled client.

The Server is composed by multiple blocks; it interfaces itself with an external network using a Standard HTTP server. This HTTP server has two main objectives: the first one is to receive the user's requests and to send them to the adaptive streaming application; the second is to deliver the data or content returned by the adaptive streaming application to the user using HTTP protocol. The Adaptive streaming application receives and analyzes the client requests, confront them with the instructions given by the control mechanisms and prepare the data to be delivered by the HTTP server (an MPD file or media segment). The control mechanisms are in charge of update, add or remove MPD profiles according to server, network or media provider constraints. They also instruct the adaptive streaming application to deliver the new MPD instead of media segments to tell users about the new service enhancing, limitation or reallocation.

The client interfaces itself with an external network using a Standard HTTP client. The main objective of the HTTP server is to open a HTTP persistent connection with a media server in order to send requests and receive data. If the data received correspond to a media segment, then the data is send to the buffer, otherwise, it is a MPD file, and this new MPD file overwrites any other previous MPD associated to the media requested. The control mechanisms at client side are in charge of signaling the player about updated MPD files, check network condition, buffer status and system resources to apply proper adaptation procedures and instruct the http client to request new media segments when necessary.

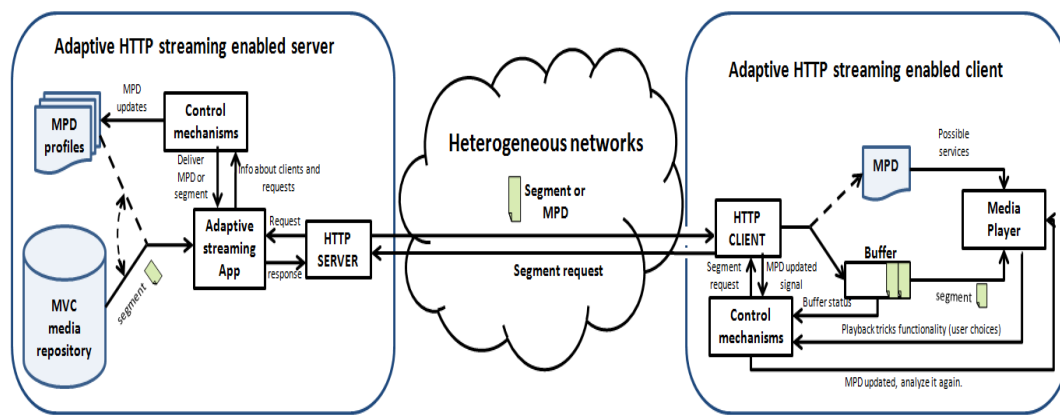


FIG 27 FRAMEWORK ARCHITECTURE

4.4 The framework's streaming process

This section describes the framework's streaming process. The main steps can be resumed as follow:

1. Client requests a media resource identified by an URL
2. Server analyze the request and send a proper MPD file
3. Client analyze the MPD and inform user on available services (i.e. available view points)
4. According to user preferences and control mechanisms the client starts downloading segments belonging to a proper representation
5. Client can automatically switch between representations or according to user's choices
6. Server can update the client's MPD

In the following subsection, each step will be detailed

4.4.1 Requesting a media resource.

The first step is requesting a media resource. Media resources are identified and located on the network by uniform resource locators (URLs). A HTTP client or a media player initiates the HTTP request (e.g. <http://MVCmedia.com/nurburgring-race>). This request establishes a Transmission Control Protocol (TCP) connection between particular ports at server and client side (typically port 80). The HTTP server listening on that port waits for the client's request message (in this case a GET message). Upon receiving the request, the server sends back a status line, such as "HTTP/1.1 200 ok", and a message of its own; the body of this message is the MPD file of the media resource. The MPD file can be generated dynamically at server side.

4.4.2 Parsing MPD file at client side and start requesting segments

At the beginning of the session the server sends the MPD file containing all the available services (views/op. points), information, segments' URLs, and metadata related to the media resource. As shown in section 4.2.2, there could be several MPD versions (profiles) for the same media resource (e.g. premium profile, base service profile or congestion's profiles) [Fig 28]. The MPD file can be updated anytime in order to enhance or limit available services (views/op. Points) depending on server needs or client category.

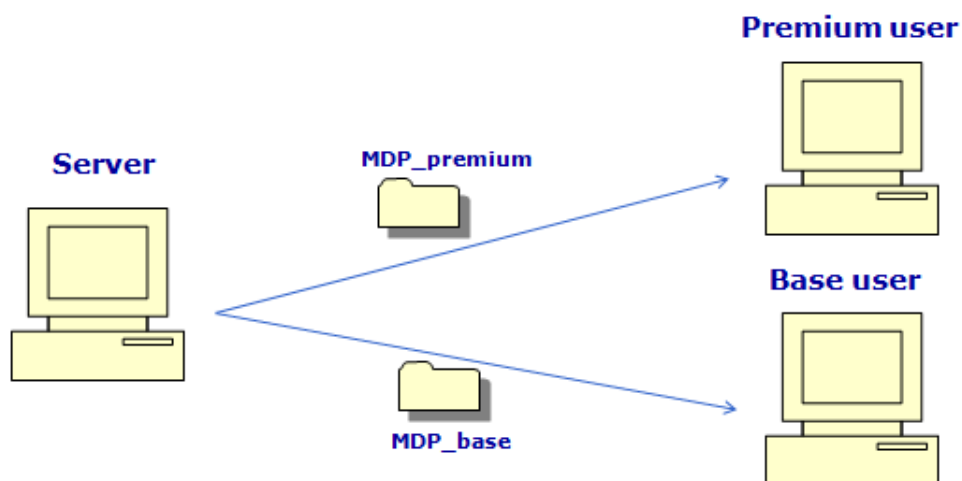


FIG 28 AT THE BEGINING OF THE SESSION, SERVER SENDS THE MPD FILE

Once downloaded, the media presentation file (MPD) is parsed by client who tells the user which services are available; at this point, it is decision of the client how to continue. E.g. the client can start the streaming with the lighter representation or can ask the user to make a choice, etc. If the user makes a choice, the client maps the choice into one of the representations contained in the MPD file. From now on, the client uses the URLs of the segments of the chosen representation to construct the HTTP requests. The media segments will be requested automatically in a linear fashion until the user or any control mechanism request something else explicitly. An example of this process is shown in [Fig 29].

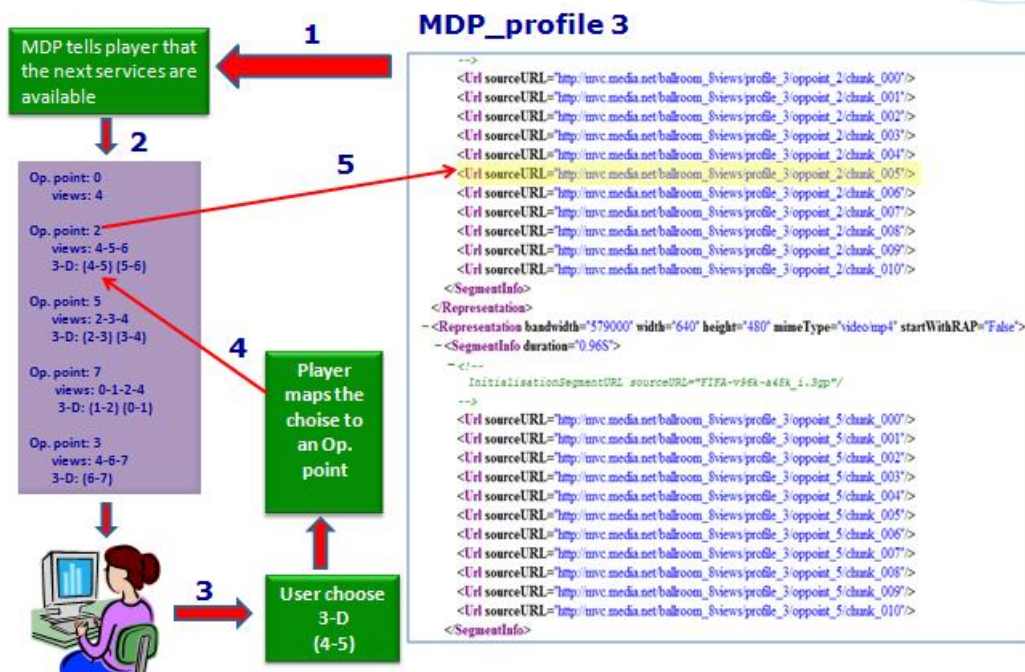


FIG 29 PARSING MPD AT CLIENT SIDE

4.4.3 Parsing HTTP requests at server side

Once the server receives the client's HTTP request for a segment, it is parsed in order to deliver, from the media container file or folder, the requested segment using classical HTTP progressive download (section 3.4.1). The information about period, representation and segment can be taken from the requested URL as shown in [Fig 30]. The adaptation process of sending segments of different representations could be done as a real time filtering instead of reading different

transmission streams from physical mediums. This possibility depends on two facts: the first one is that the coding technology and file format consent on the fly filtering of the main stream to create sub-streams, and the second is the trade-off between complexity of handling many spliced sub-streams and resources to do real time filtering. [Fig 30] shows a server that uses real time filtering as approach.

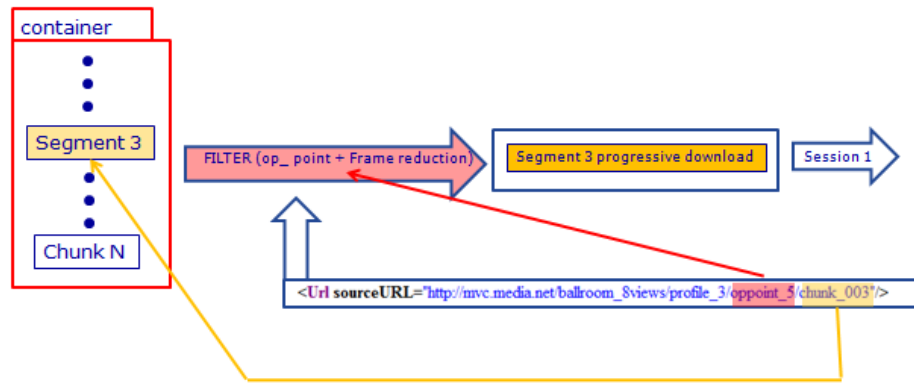


FIG 30 PARSING HTTP REQUEST AT SERVER SIDE

To perform real time adaptation of the H.264/MVC bitstream at the server side, the server can use the view scalability SEI Message contained in MVC bitstreams, which contains information about the characteristics of the operation points and their dependencies. When present, this view scalability SEI message appears in an IDR access unit. The semantics of the message are valid until the next view scalability SEI message. Using the client's request the server can use this SEI message to discard those NAL units that do not belong to the requested op. point. An adaptation filter for H.264/MVC streams is deeply explained in (15). An example of filtering adaptation of the H.264/MVC bitstream shown in [Fig 15] is depicted in [Fig 31].

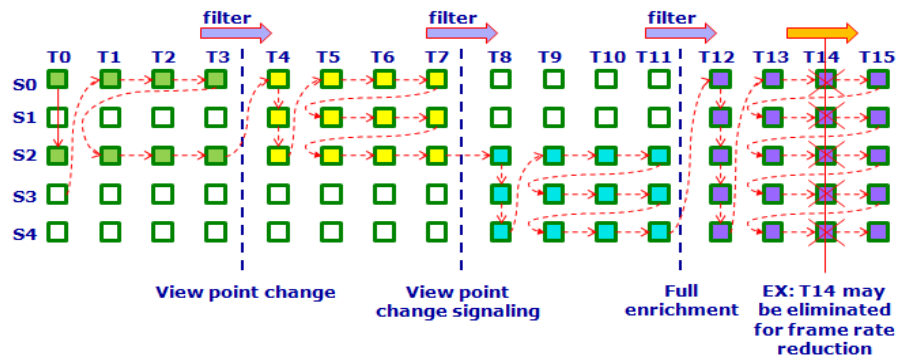


FIG 31 MVC DATA CHAIN FILTERING

4.4.4 Segments requests and delivery with HTTP/1.1

HTTP can transport arbitrary binary data. HTTP/1.1 let resources that are generated dynamically (generated on the fly, created as output from scripts or programs based on user input), to be transferred even if their exact size cannot be determined in advance. Also, HTTP/1.1 chunked transfer allows the server to send out parts of the response as soon as they become available from the application. These two features are important for the mentioned real time filtering at client side.

HTTP/1.1 uses persistent connections that allow multiple request and responses to be sent over a single TCP connection, meaning this that, at any point of the transmission of a segment, the client can ask for another operation point or representation. The server can prepare itself to transmit the next segment with the new client's choice. [Fig 32] shows two different schemes for request and deliver segments with HTTP/1.1. The right scheme request segments once they have been downloaded; the left scheme make the requests inside the time frame of segment download process; this could help to improve response time and delay.

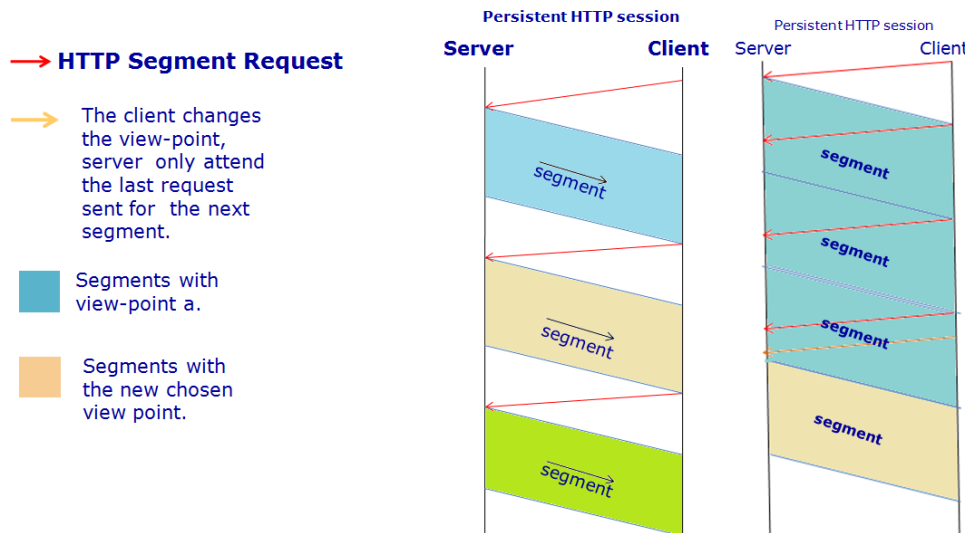


FIG 32 TWO DIFFERENT SCHEMES FOR REQUEST OF SEGMENTS WITH HTTP/1.1

4.4.5 MPD updates

Section 4.2.2 introduced the concept of MPD profiles which, in few words, explain how a media resource can have more than one MPD file, and how this interesting property combined with the fact that the client's MPD file can be updated inside a streaming session (following the 3GPP standard's guideline introduced in section 3.6.4) will open the doors to new server's mechanisms of control and network scalability.

The process of change, add or remove a MPD of the MPD profiles and update a client's MPD file are asynchronous process. First, a control mechanism (intern or extern to the server) update or add an MPD profile at server side due to for example bandwidth policies, or available resources. Then, the Streaming server sends with the next segment request the new MPD file. It is work of the client to detect that a new MPD file has arrived instead of a media segment, analyze the new MPD, then immediately choose a new representation (or the older one if it is available in the new MPD) and ask again for the previous segment.

4.5 Scalability and adaptivity features

4.5.1 On the fly adaptation at server side

On the fly adaptation at server side is useful when media servers have different user's categories, bandwidth constraints or limited resources. It is done by updating the MPD files of the clients connected to the server. The new updated MPD file enhance or limit the services offered for a certain media resource. It is work of the control mechanisms to update the MPD files at server side, then the adaptive streaming application tells the HTTP server to transmit the new MPD instead of the next requested segment. Imagine a situation where a content provider is streaming a live event offering MPDs with representations up to 3-D eight view points to premium users and 3-D one view-point with advertisings to base users, and at some time many base users connect to the content provider requesting the event. Such situation jeopardize the premium users with possible lags; then, the content provider updates the MPD files of the base users with an MPD that offers up to 1 view-point at VGA resolution to assure QoS of premium users. The process is shown in [Fig 33].

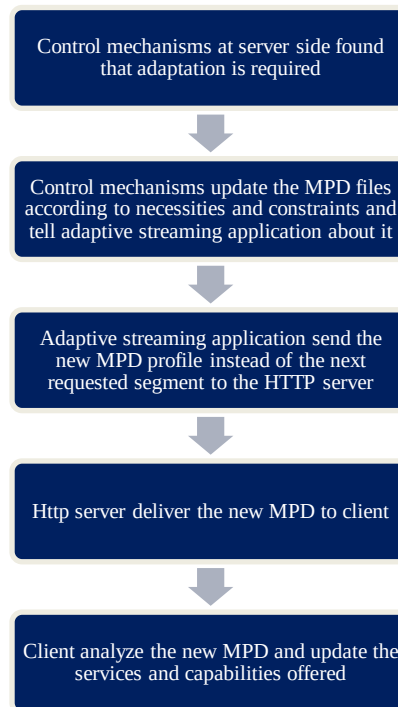


FIG 33 ON THE FLY ADAPTATION AT SERVER SIDE

4.5.2 *On the fly traffic redistribution*

Another way to enhance QoS and network resources is by means of traffic redistribution. An easy way is by changing the URL of the media segments contained in the MPD files, and then updating the clients' MPD. Seamless switching to new servers are now made by clients, who just change the base URL of the HTTP requests. An external control mechanism at CDN side could check periodically the load status of the servers and efficiently distribute traffic between them. The process follows [Fig 33], but, at the last step, instead of updating services, the URLs of the segments are changed. For example, in [Fig 34] a lot of users were requesting a media resource to sever 1; the main traffic control of the CDN give instructions to server 1 to redistribute the traffic between server 2 and 3. This is done by updating the MPD of some of the clients with URLs pointing to server 2 and the other users to server 3.

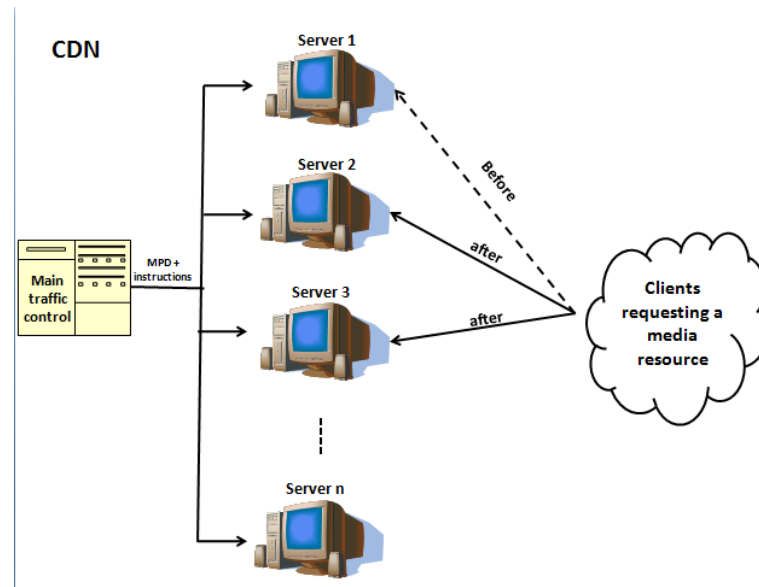


FIG 34 TRAFFIC REDISTRIBUTION BETWEEN CDN SERVERS

4.5.3 Playback tricks and preferred 2D/3-D view-points

Assume that a client attempts to seek to a specific presentation time, tp , in a Representation with start time 0. Based on the MPD, the client has access to the media segment start time and media segment URI of each segment in the representation. The segment most likely to contain the desired media time tp is obtained as the maximum segment index i , for which the start time is smaller or equal to the presentation time tp .

$$segment_index = \max \{ i \mid \text{MediaSegment}[i].\text{StartTime} \leq tp - \text{Period.start} \}.$$

Note that timing information in the MPD may be approximate due to issues related to placement of Random Access Points, alignment of media tracks and media timing drift. As a result, the segment identified by the procedure above may begin at a time slightly after tp and the media data for presentation time tp may be in the previous media segment. In case of seeking, either the seek time may be updated to equal the first sample time of the retrieved file, or the preceeding file may be retrieved instead. However, note that during continuous playout, including cases where there is a switch between alternative versions, the media data for the time between tp and the start of the retrieved segment is always available. By

continuously using partial HTTP requests, only the relevant parts of the media segment may be accessed for improved user experience and low start-up delays.

The client may also pause or stop a media presentation. In this case client simply stops requesting media segments or parts thereof. To resume, the client sends requests to media segments, starting with the next or previous segments after the last requested segments, or could ask the same segment with a partial HTTP requests and access the previous RAP. If buffering is present, client could also finish the current segment download, but stop the flow of data to the player, until the media is run again, the flow start again and new segments are required when needed. The client may use multiple representations to support trick mode behavior.

Based on updated information during an ongoing media presentation, a client may decide and desire to switch representations. Switching to a “new” representation is equivalent to tuning in or seeking to the new representation. Once switching is desired, the client should seek to a RAP in the “new” representation at a desired presentation time tp later than the current presentation time. Presenting the “old” representation up to the RAP in the “new” representation enables seamless switching.

4.5.4 Automatic client adaptation

Client adaptation is triggered by client’s side control mechanisms. Is simply done by changing the current representation being requested into other that fits better clients’ constraints. For example, user can start with the lightest MPD’s representation in order to minimize the starting delay. If certain conditions are met, the client automatically changes to a higher quality/more services representation. In case of buffer starvation client can switch to a lighter representation. Control mechanism can vary in nature, from checking constrains on energy consumption to requirements on resolution, bandwidth and resources. Control mechanisms are a client side implementation and choice, and remain a feature for product differentiation. In this thesis work a control mechanism based on an optimized buffer scheme for adaptive HTTP streaming with bandwidth rate control was implemented; it is mentioned in chapter 6 and a demo showing its performance is introduced in chapter 7.

5

Efficient MVC Coding Structures for the Adaptive HTTP Streaming Framework

As mentioned in previous chapters, the temporal plus inter-view prediction is an important feature of MVC coding, but, many different schemes can be considered for prediction, exploiting both temporal and inter-view statistical redundancies. It must be considered that there are many factors influencing the coding performances, for example, camera distances, frame rate, and the complexity of content (27). The results provided by (10) showed that exploiting temporal redundancies was better than exploiting spatial redundancies (and this last one was better than other combined solutions). The most used coding scheme, shown in [Fig 14], uses a prediction structure with hierarchical B pictures for each view. The first view to be encoded is called “base-view”, and does not exploit inter-view prediction. To allow synchronization, I-frames (anchor frames) start each GOP. If a frame in one view is an anchor frame, then all frames with the same temporal index in other views shall be anchor pictures. This temporal index is called “anchor temporal axis”. This chapter aims to introduce a coding structure that will help to save bandwidth when used for adaptive HTTP streaming.

5.1 Efficient coding structures for bandwidth saving

This thesis work proposes a multi-view coding structure with hierarchical B pictures for both temporal and inter-view prediction. The proposed structure is shown in [Fig 35].

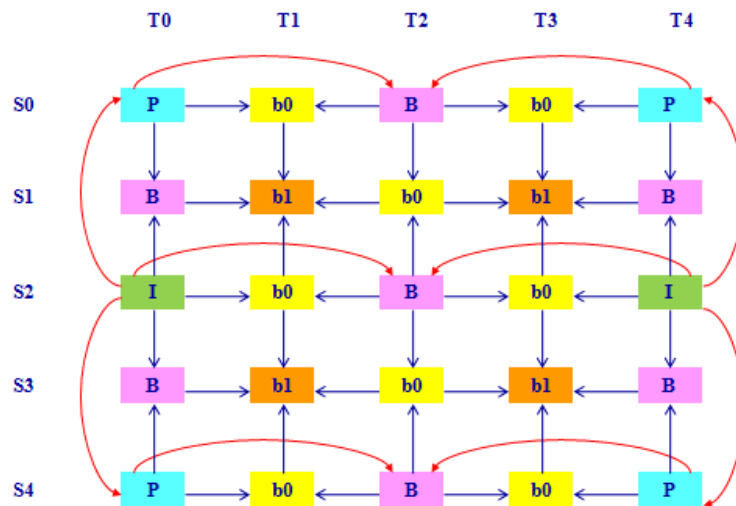


FIG 35 PROPOSED MULTIVIEW CODING STRUCTURE

The proposal hypothesis supports itself in the basis that most camera arrangements are symmetrical respect a certain central camera. This can be

exploited by the streaming framework since any requested view-point must belong to one of the sides of the camera arrangements, i.e. one side is completely filtered.

From a streaming service point of view, this structure can save up to half of the bandwidth used by the classical structure since users are likely to stay and keep requesting a certain view-point. For a free view-point service, it is the same, as a smooth navigation through a scene uses adjacent cameras, i.e. the cameras from the same side of the arrangement. [Fig 36] shows the available operations points for the proposed structure and [Fig 37] shows adaptation at server side using real time filtering of the MVC elementary stream.

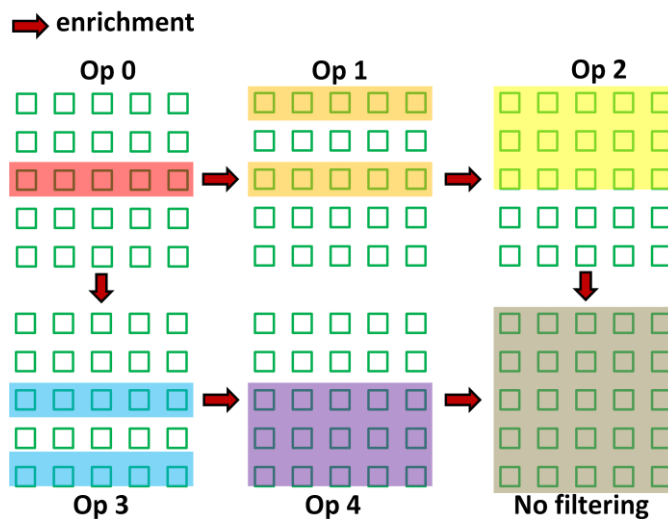


FIG 36 OP POINTS AND POSSIBLE ENRICHMENTS FOR THE ENHANCED STRUCTURE

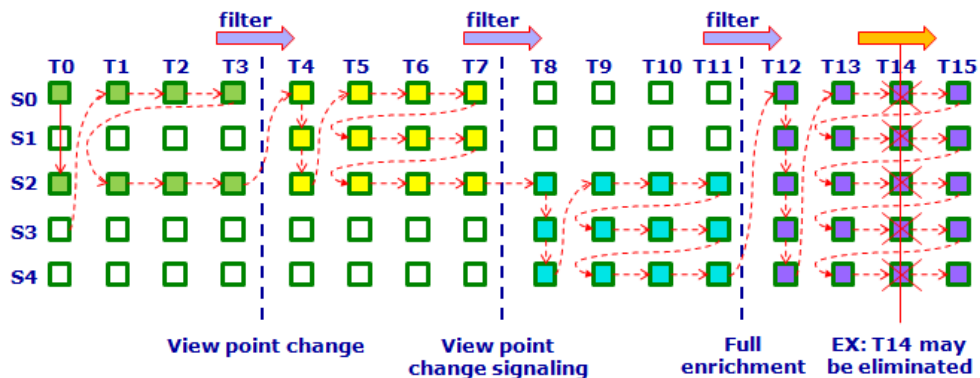


FIG 37 MVC STREAM DATA CHAIN FILTERED IN REAL TIME

The proposed structure could lead to reasoning that the best solution could be to eliminate any interview dependency between the views to save the most bandwidth possible. But this is far from being true. First of all, 3D view-points services and free view-point services use two or more cameras. Particularly, for free view-point several cameras can be used to offer smother navigation trough a scene, requiring, thus, correlated coding structures to reach maximum compression ratios. Second, MVC offers the best compression ratios between coding schemes and structures as shown in the next comparison of a set of four encoding methods (27):

1. **MVC**, i.e. encoding all views together, exploiting temporal and inter-view redundancies by means of hierarchical B-pictures structure and inter-view prediction.
2. **H.264 Anchor**, i.e. encoding each view independently, using the classical IBBPBBP prediction scheme.
3. **H.264 Simulcast**, i.e. encoding each view independently, using the hierarchical B-pictures structure
4. **MVC Simulcast**, i.e. the same as H.264 Simulcast, but using MVC and considering each view as if it was the base-view. Note that “encoding each view independently” means “without any references from the other views”, so that just the temporal prediction is used.

In [Fig 38]it is possible to notice the great increase of quality (at the same bitrate) that stems from the use of the novel prediction scheme based on hierarchical B scheme of H.264 Anchor, equal to about 2.21db (on average). Instead, the quality gain versus MVC Simulcast and H.264 Simulcast is respectively equal to about 1.3db and 1.5db.

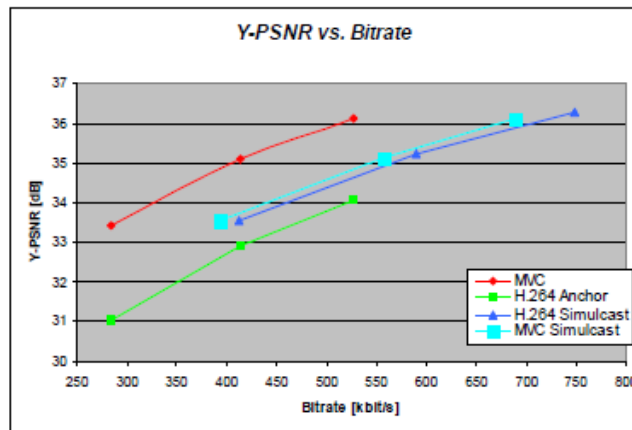


Fig 38 Comparison between coding methods.

6

Framework Implementation

An adaptive HTTP streaming enabled server and an adaptive HTTP streaming enabled client were implemented. The implementation follows the architecture depicted in [Fig 27].

At client side VideoLAN media player (VLC) (3) has been used and extended with additional tools and modules to provide the streaming capabilities for the client. These extensions are explained in section 6.3

At server side, NULL HTTPD software is used as HTTP server (open source web server). This HTTP server has been extended into an adaptive HTTP streaming server using a common gateway interface (CGI) program. All major commercial and open source HTTP servers allow CGI programs. A control mechanism has also been implemented, which is a program that updates the MPD repository with new MPDs according to server constraints.

The operating system chosen for the implementation at server and client side was Ubuntu 10.0 64 bits (a Linux-based distribution). The main programming languages used were C and C++. A Parallel implementation was done in the same operating system but using the 32 bits version.

Section 6.1 discusses the coding and formatting of media resources; section 6.2 discusses the framework implementation at server side, and finally section 6.2 discusses the framework implementation at client side. The whole implementation is used in chapter 7 to make the test bed for the tests presented in the same chapter.

6.1 Media Resources

As explained in previous chapters, the client request media segments belonging to different representations of the MPD file, and somehow the server has to generate or extract these segments from a folder or container file.

As explained in chapter 4, no file format is used in the implementation, but, the framework remains flexible about them. The implemented framework streams segments of raw H.264/MVC. There is a file for every single segment of every possible representation. It should be mentioned that it is possible to have just the files for the segments containing the elementary stream, and perform a filtering at server side to extract substream for lighter representations (see section 4.3.3). This section explains how to create these segments using the JMVC software from RAW Multi-View sequences.

6.1.1 Creating H.264/MVC streams with JMVC 8.0

The JMVC (Joint Multiview Video Coding) software (27) is the reference software for the Multi-view Video Coding (MVC) project of the Joint Video Team (JVT) of the ISO/IEC Moving Pictures Experts Group (MPEG) and the ITU-T Video Coding Experts Group (VCEG). Since the MVC project is still under development, the JMVC Software is also under development and it changes frequently. The JMVC software is written in C++ and is provided as source code. [TABLE 3] shows the executables provided by the JMVC software

<i>executable</i>	<i>description</i>
<i>DownConvertStatic</i>	<i>resampler</i> The resampler can be used for spatial/temporal resampling (up-sampling or down-sampling) of video sequences. It is used in this thesis work to fragment the raw video sequences into segments of X frames each one.
<i>H264AVCEncoderLibTestStatic</i>	<i>AVC/MVC encoder</i> The encoder can be used for generating single-view (MVC) or multiview (MVC) bitstreams.
<i>H264AVCDecoderLibTestStatic</i>	<i>MVC decoder</i> The decoder can be used for decoding AVC or MVC bitstreams and reconstructing raw video sequences.
<i>MVCBitStreamAssembler</i>	<i>MVC assembler</i> The bitstream assembler should be executed after running the encoder to generate a bitstream containing all the encoded views. The assembling of the views is done in time-first order.
<i>MVCBitStreamExtractor</i>	<i>MVC extractor</i> The bitstream extractor can be used to extract sub-streams of an MVC stream. The substreams represent streams with a reduced number of views.

TABLE 3 EXECUTABLES PROVIDED BY THE JMVC SOFTWARE

A Multi-View video sequence firstly comes as a set of raw video files (uncompressed), each one corresponding to a different view-point or camera. Every raw file is coded into a H.264/MVC stream using *AVC/MVC encoder*. After running the encoder, the next step is to use the *MVC assembler* to generate a bitstream containing all the encoded views. The assembling of the views is done in time-first order. The last step is to run the *MVC extractor* to generate the different substreams. The substreams represent streams with a reduced number of

views. Corresponding to different operation points, these substreams also correspond to the different level of number of views scalability, present in the representations of a MPD file. The process is depicted in [Fig 39].

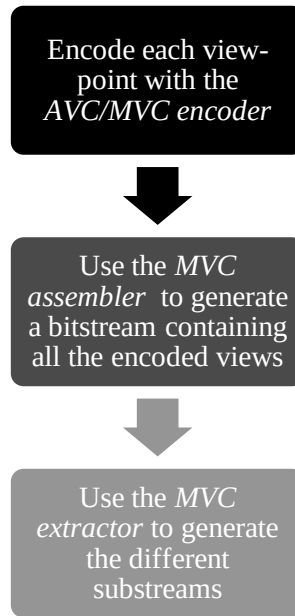


FIG 39 CREATING H.264/MVC STREAMS WITH JMVC 8.0

The basic encoder call is illustrated in [CALL 1]. At this, *mcfg* represents the filename of the main configuration file, deeper explained in (27). The main configuration file shall be specified for each encoder call. The parameter *view_id* represents the view that should be coded. The encoder should be run for each view that is to be encoded.

```
H264AVCEncoderLibTestStatic.exe -vf <mcfg> <view_id>
```

CALL 1 BASIC JMVC ENCODER CALL

The MVCBitStreamAssembler tool, as mentioned before, is used to assemble the individual bitstreams that represent the different views into a single bitstream. A configuration file is used to specify in what order the bitstreams need to be assembled. This order should be the view coding order. The assembler assembles the bitstream in the view coding order such that the pictures at a certain time instant are together. The basic MVCBitStreamAssembler call is illustrated in [CALL 2]. Configuration file examples can be found in (27).

```
H264AVCEncoderLibTestStatic.exe -vf <mcfg> <view_id>
```

CALL 2 BASIC JMVC BITSTREAM ASSEMBLER CALL

The bitstream extractor can be used to extract sub-streams of an MVC stream. The usage of the bitstream extractor is illustrated in [CALL 3]

```
>MVCBitStreamExtractorStatic input.264 output.264 -op x
```

CALL 3 CALL FOR THE EXTRACTION OF A SPECIFIED OPERATION POINT

The parameter *Input* specifies the filename for the input bitstream (global bitstream). The parameter *Output* specifies the filename for the output bitstream, which generally represents a sub-stream of the input bitstream as specified by the additional command line parameters. The option *-op* is used to specify the operation point ID that should be extracted. The output is all the packets that will be used in the decoding of the specified operation point id.

6.1.2 Generation of segments

As mentioned in section 3.6.3 and following the guideline in (16), segments must be self contained; no dependencies on other segments are supposed to exist. With this consideration in mind, the first and last GOP of a segment must not have dependencies on GOPs of other segments. A way of create these segments assuring this constraint is using the *JMVC resampler* to fragment the raw video sequences with the desired number of frames and then apply to each segment the process described in 6.1.1.

6.2 Implementation of an adaptive HTTP streaming enabled server

The implemented Server follows the architecture illustrated in [Fig 40] and explained in (section 4.3). The implementation of the modules or parts of the mentioned architecture are explained in the following subsections.

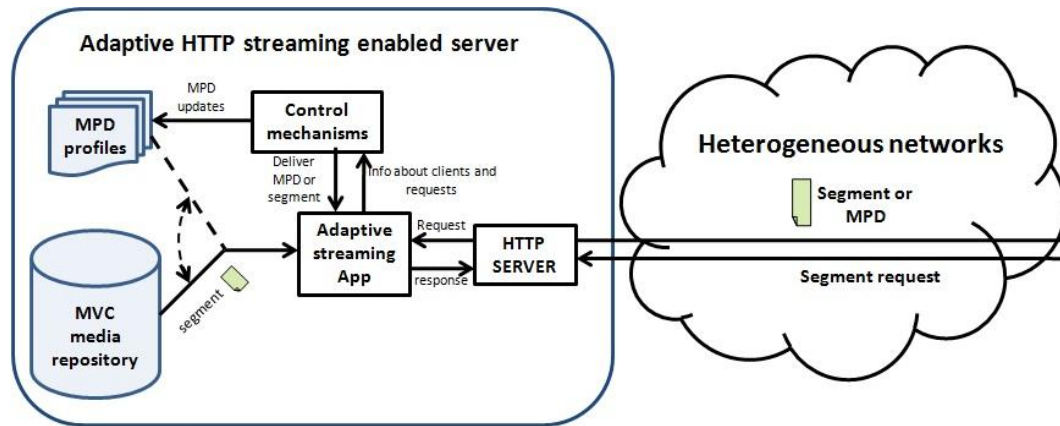


FIG 40 SERVER SIDE ARCHITECTURE

6.2.1 The HTTP server: NULL HTTPD

A HTTP server is a computer program that delivers content, using the Hypertext Transfer Protocol (HTTP). A client initiates communication by making a request for a specific resource using HTTP and the server responds with the content of that resource, or an error message if unable to do so. This implementation uses Null httpd (28), a tiny web server which is designed to be very small, simple, multithreaded, and available for Linux and Windows. Null httpd is a free software/open source, and is distributed under the GNU General Public License (GPL).

Null httpd is configured by placing directives in plain text configuration files. The main configuration file is usually called `httpd.cfg`. [TABLE 4] shows some important directives settings for Null httpd.

<code>SERVER BASE DIR = "/usr/local/httpd"</code>	Base directory
<code>SERVER BIN DIR = "/usr/local/httpd/bin"</code>	binary directory
<code>SERVER CGI DIR = "/usr/local/httpd/cgi"</code>	CGI and extension modules
<code>SERVER ETC DIR = "/usr/local/httpd/etc"</code>	Configuration files
<code>SERVER HTTP DIR = "/usr/local/httpd/html"</code>	HTML documents
<code>SERVER LOGLEVEL = "1"</code>	Verbosity for log file
<code>SERVER HOSTNAME = "any"</code>	Host name
<code>SERVER PORT = "80"</code>	Listening port
<code>SERVER MAXCONN = "50"</code>	Max number of users
<code>SERVER MAXIDLE = "120"</code>	Max time idle for persistent connections

TABLE 4 CONFIGURATION SETTINGS FOR NULL HTTPD

In most cases the server cannot itself process the data. Most standard HTTP servers programs are designed only to serve out documents, and are not designed to process data sent from a client. Therefore, to do server-side processing of the data sent from a client, it is necessary to extend the http server capabilities with a CGI program. This should be located in the CGI directory by default. It is also need to be mentioned that the maximum idle time for persistent connections must be a few times greater than the media segment's length due to the buffering systems at client side. The client could ask for next segments just after previous downloaded segments have been displayed. If the maximum idle time is smaller than the segment length, the server may close the connection with the client, requiring to open a new one with the next request.

6.2.2 *Common Gateway Interface (CGI)*

The HTTP server can interface itself with a secondary programs to extend its capacities. This program can be in charge of process the data sent by the client, prepare new data and return it to the client. Such secondary programs are called gateway programs, because they act as a gateway between the Web and other resources on the HTTP server machine, such as databases (29).

An example of this flow of information is shown in [Fig 41]. The arrows show the flow of data, while the small ovals straddling the lines show the protocols and mechanisms by which the data are communicated from client (browser)-to-server-to-gateway and back again. These are called *gateway* programs because they generally act as gateways between the World Wide Web and server-side resources such as databases, feedback forms, clickable imagemaps, and so on.

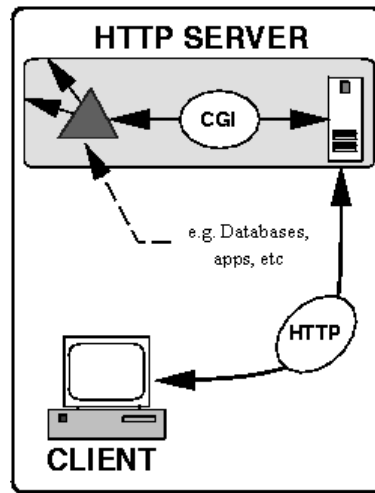


FIG 41 COMMON GATEWAY INTERFACE

CGI (Common Gateway Interface) is a standard for interfacing external applications with Web servers. Usually, both free and commercial Web servers support it. CGI programs, it can be written in programming languages like C, PERL, or scripting languages like PHP, etc. They are understood by modules designed to run CGI programs implemented in the HTTP server applications.

A CGI application is launched by the HTTP server each time a HTTP request like the one depicted in [CALL 4] arrive.

```
>http://server-adress/cgi-bin/name-cgi?text
```

CALL 4 URL TO REQUEST A CGI SERVICE

The http server receives this request and understands that the CGI *name-cgi* needs to be launched. Then, for example the CGI could use the content of *text* which can be found in an environmental variable as a parameter to process data. Usually, the CGI returns this data to the HTTP server using the standard output. [Fig 42] illustrate the process.

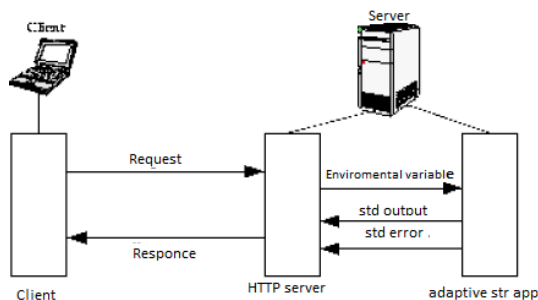


FIG 42 CGI COMMUNICATION WITH HTTP SERVER

6.2.3 The adaptive streaming extension for HTTP server

The adaptive streaming application is implemented by extending the HTTP server with a CGI. One of the main objectives of the adaptive streaming application is to send the media segments to the HTTP server using the standard output. The HTTP server sends them to the client using HTTP protocol. If the segments are just files, any HTTP server could send the requested segments by itself, without the need for a CGI program. There are several valid reasons to use this CGI extension. For one, it let store the media segments in a modern RDBMS (*Relational Database Management System*) that supports BLOBs (*Binary Large Objects*). To get the segments to the user, it is necessary to retrieve them from the database server first, which no current HTTP server is able to do.

Another reason is that the segment files do not have to be in the public WWW area, so it is possible to send any file in the file system or other server the CGI have access to. Other applications are, for example, sending segments that are generated on the fly, by using the filters mentioned in 4.4.3, and implement the mechanism of control that decides to send one of several possible segments or an MPD file instead of a media segment. The functionalities of the adaptive streaming application are mentioned in section 4.3.

The adaptive streaming application was implemented using C. The binary file called *nurburgring* is located in the CGI folder in the main Null httpd folder. Each time a HTTP request like the one illustrated in [CALL 5] arrives, the process illustrated in [Fig 43] is run. It is job of a third control mechanism to update the MPD according to servers' constraints or requirements. If the CGI extension checks that the MPD has been updated, it sends a MPD to the http server. Otherwise, it analyzes the request and looks for the segment x in order to send it to the http server, using the standard output interface in both cases.

```
>http://server-adress/cgi-bin/nurburgring?segment=x
```

CALL 5 URL TO REQUEST A SEGMENT TO THE ADAPTIVE STREAMING EXTENSION

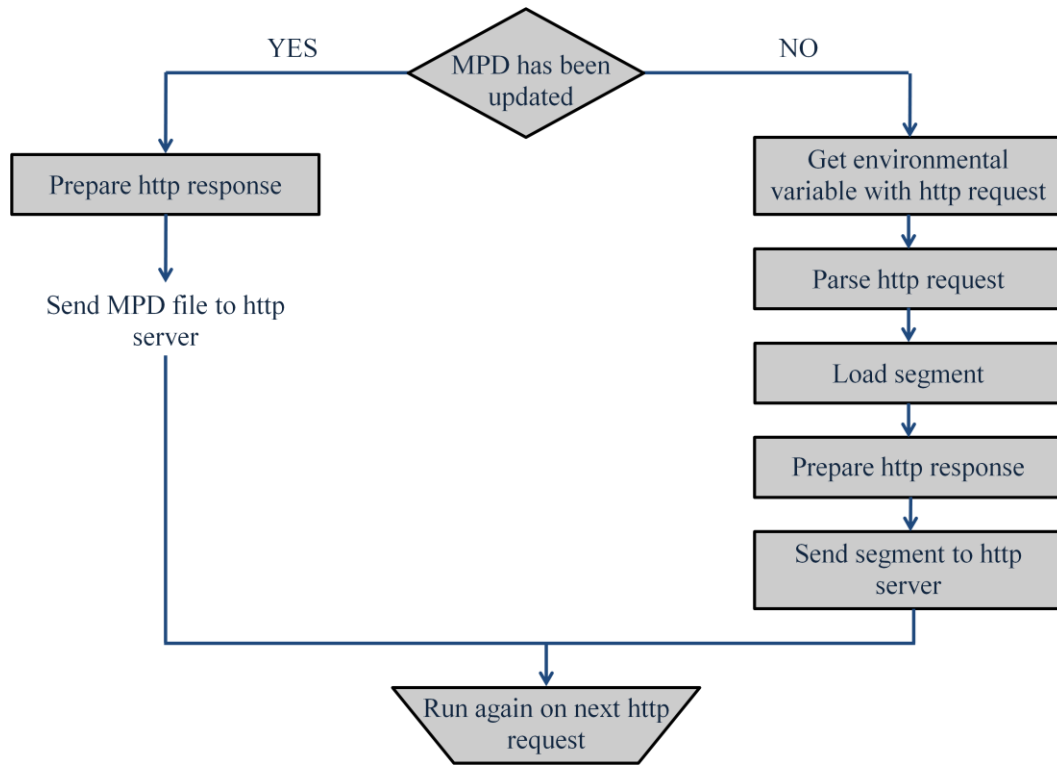


FIG 43 ADAPTIVE STREAMING APPLICATION FLOW CHART FOR EACH SEGMENT REQUEST

6.2.4 Control mechanisms by third party applications

The control mechanisms implemented in this thesis work are simple scripts that update the MPD file contained in the HTML documents folder of the HTTP server with another contained in a MPD repository [Fig 44]. Each MPD in the repository correspond to a different MPD profile (4.2.2), which eventually, will trigger adaptation, or traffic redistribution processes. Once updated, the Adaptive streaming application will send it to user with the next segment request.

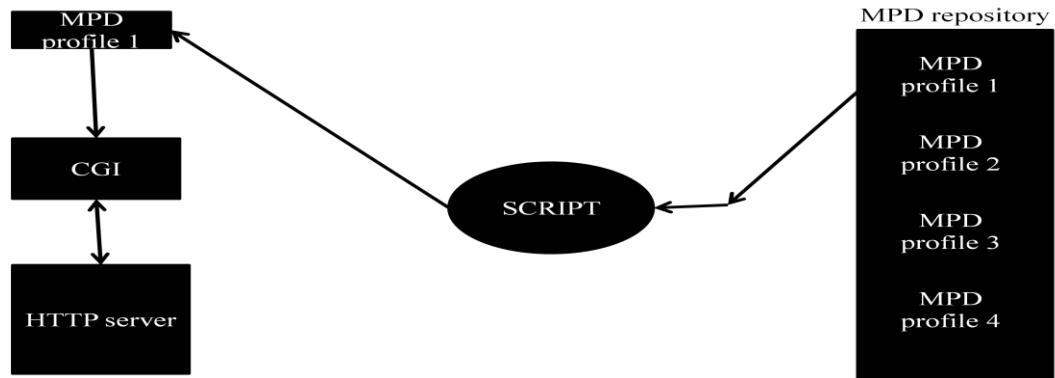


FIG 44 SCRIPT FOR UPDATE THE MPD AT SERVER SIDE

6.3 Implementing the streaming framework at client side

At client side, VLC media player (3) version 0.8.6 has been used and extended with additional tools and modules in order to provide all the streaming capabilities, particularly:

1. A new codec plug-in to interface VLC with JMVC 8.0 in order to support decoding of H.264/MVC content.
2. A custom HTTP streaming client. This module implements handling of media segments and MPD file, buffer management and control mechanisms based on rate control for automatic adaptation.
3. A custom demuxer. This module takes information about frame rate, resolution, etc. from the MPD and send it to the decoder. The demuxer is also in charge of format the media segments, in a way the decoder plug-in understands.

The implemented client automatically recognize if the data received correspond to a media segment or to a MPD file. The control mechanisms at client side are in charge of inform the player user interface about updated MPD files. Furthermore it checks network condition, buffer status and system resources in order to apply proper adaptation procedures.

The flow of data is shown in [Fig 45] and it consists of the following steps:

1. A MVC segment or a MPD arrives to the custom HTTP client.

- a. If a MPD arrives, it overwrites any previous MPD, and then, it is analyzed by the HTTP custom client and the graphic user interface is updated with the new services. The previous requested segment is requested again.
 - b. If a MVC segment arrives, the custom HTTP server sends it to the buffer, estimates network bandwidth and choose the next segment to be requested from a proper representation contained in MPD (adaptation process). This request will be done in a precise time to keep the buffer size constant and equal to a predefined number of segments.
2. The custom Demuxer access the buffer where the segments are and starts to take set of pictures belonging to the same time instant (pictures from the multiple views), and these are arranged as chunks. A chunk is supplied to the Decoder plug-in. If the MPD has been updated, the Demuxer loads the information needed by the decoder from the MPD and sends it to it.
3. The decoder plug-in calls the decoder functions of the reference JMVC software. These functions return a set of temporally synchronized decoded pictures belonging to the multiple views.
4. Post-processing is performed on the decoded pictures to identify the views needed to create the viewpoint to be displayed and to create the anaglyph picture if needed for the 3D effect. After the post-processing the viewpoint requested by the user is displayed.

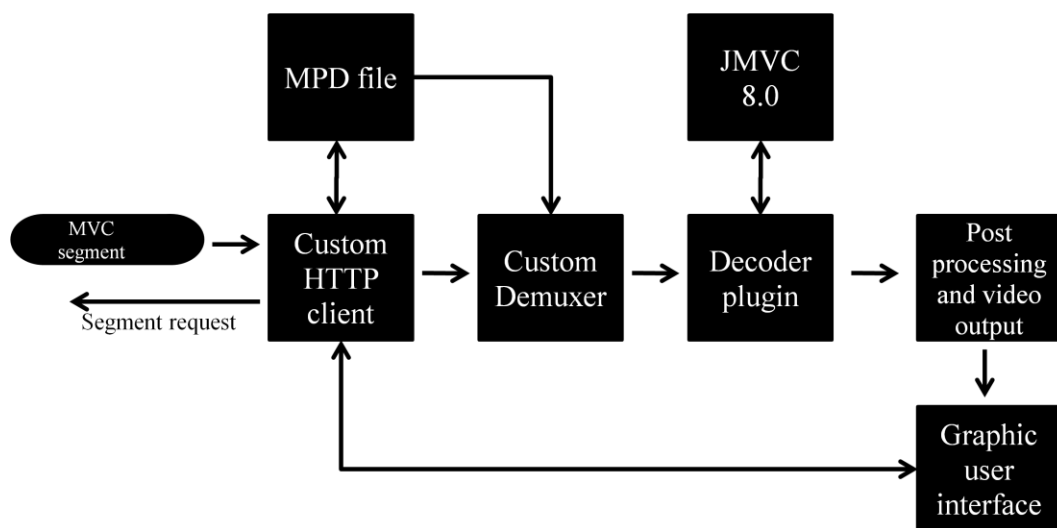


FIG 45 FLOW OF DATA IN VLC

The whole streaming client is contained in the *dev_jeh_32bit_jmvc* directory. In this folder there are two shell files: *build all.sh* and *clean all.sh*. The first is used to build VLC and all the plug-ins and codecs, and the second to clean the streaming system. *Build all.sh* allows configuring from terminal some parameters of the system. The conciliated parameters are shown in [TABLE 5]. A troubleshooting guide for the compilation process of the client can be found in Appendix A: Troubleshooting for the client's compilation process.

Parameter	Value
parallelcomp=	3
debug=	"n"
osrel=	"Gentoo"
machine=	"Intel"
mmx=	"y"
encodercomp=	"n"
pcinstalldir=	"\${cur_dir}/binaries/Intel"
boardinstalldir=	"\${cur_dir}/binaries/7100"
useHHIcodec=	"n"
usePVcodec=	"n"
mpeg21tools=	"n"
useMDCcodec=	"n"
usejmvccodec=	"y"
useSDPapproach=	"n"

TABLE 5 CONCILIATED PARAMETES FOR THE BUILDALL.SH SCRIPT

6.3.1 VLC 0.8.6

VLC Media Player is an open source, free portable media player written by the VideoLAN project. It is licensed under GNU General Public License version 2 and its source code is available for free at the VideoLAN website (3). The VLC media player is available for a wide range of Operating Systems such as Windows, Linux, Solaris, Mac OS X etc. and is widely used, being downloaded over 100 million times.

VLC has a highly modular design which makes it easier to include modules for new file formats, codecs or streaming methods. This principle also stretches to other areas and there is a wide selection of interfaces, video and audio outputs, controls, and audio and video filter modules. [TABLE 6] lists the VLC source tree.

Directory Name	Directory Explanation
bindings	Java, CIL and Python bindings
doc	Documentation about VLC
extras	Code style editor and macro
include	Header files for VLC
libs	Contains the loader (win32 dmo codec loading on Linux) and a SRTP library
lxdialog	Make menuconfig files
m4	Macro files needed for automake and autoconf
modules	Contains the VLC modules. It's the most important folder
po	Language translation files
projects	Projects based on libvlc, Mozilla plugin, ActiveX plugin and Mac OS X Framework
share	Icons, scripts to make VLC the default player etc.
src	Contains the VLC sources files.
test	scripts to see if everything is OK

TABLE 6 VLC SOURCE TREE

The directory “src” contains the core files of VLC. The directory “modules” contains the modules that extend VLC capabilities. The modules can communicate between them by buffers, FIFOs and objects usually defined in “src”. It is a good programmer practice for a plug-in developer not to modify the “src” directory, in order to keep rules and compatibility for other modules.

The principal modules of the directory “modules” is described in [TABLE 7]. The custom HTTP client, the custom Demuxer and the MVC codec plug-in are located in the folders “access”, “Demux” and “Codec” respectively.

Directory Name	Description	Directory Name	Description
Access (/cdda, /dvb, /mms, /rtsp)	Protocols to access streams through network (http, Ftp, Tcp, Udp etc.), access to physical media like CD's and DVD's	Interface	Code for user interaction such as key presses and device ejection.
Access - filter		Playlist	Manages playlist interaction such as stop, play, next, or random playback.
Access - output		Input	Opens an input module, reads packets, parses them and passes reconstituted elementary streams to the decoder(s).
Audio - filter	Various audio filters like decoders, equalizers, converters.	Video_output	Initializes the video display, gets all pictures and sub-pictures (i.e. subtitles) from the decoder(s), optionally converts them to another format (such as YUV to RGB), and displays them.
audio-mixer		Stream_output	Manages the functionality of video streaming.
Audio - output		Misc	Miscellaneous utilities used in other parts of “libvlc”, such as the thread system, the message queue, CPU detection, the object lookup system, or platform-specific code.
Codec (/FFmpeg etc.)	Includes various codecs, notably FFmpeg which is used for encoding and decoding various formats		
Control	Various interfaces to control the player: gestures, hotkeys, remote control and telnet		
Demux (/asf, /avi, /mp4 etc.)	Various demuxers		
Gui (/skins2, /wxwidgets etc.)	Graphical User Interface		
Misc (/playlist, /xml etc.)			
Mux (/mpeg, /rtsp)	Various muxers		
Packetizer	Packetizers for H264/AVC and MPEG 4 audio and video streams.		
Stream_out	Module for streaming		
Video - chroma	Image conversions such as YUV to RGB		
Video-output (/direct, /x11 etc.)	Video Output module		
Visualization			

TABLE 7 VLC MODULES DIRECTORY

6.3.2 Decoder Plug-In To Interface VLC with JMVC 8.0

In order to decode the chunk supplied from the custom Demuxer (a set of frames belonging to different points of view but same time instant) the decoder module of VLC Media Player needs to call the functions from the H.264/MVC decoder of the JMVC 8.0, which returns images for displaying. An earlier version of a plug-in that interface VLC with earlier versions of JMVC was available, and it was needed to modify some of the existing functions and structures to leave this plug-in compatible with JMVC 8.0. Particularly, there were added some inexistent fields related to the number of views. The plug-in takes a de-packetized chunk and calls the decoder function which returns a set of temporally synchronized decoded pictures belonging to the multiple views. The decoder plug-in is implemented in the file “/STMH264MVCcodec/src/mvcdecoder.cpp”. [TABLE 8] lists some of the important structures and methods of the decoder plug-in.

Structure	Description
image_st	This structure stores a decoded picture.
Method	Description
H264MVCOpenDecoder()	This method is used to initialise the decoder.
H264MVCDecode()	This method is used to decode a block.
DisplayImage()	This method creates an image_st structure for the decoded picture.
H264MVCCloseDecoder()	This method closes the decoder and frees the data structures used by the decoder.

TABLE 8 IMPORTANT STRUCTURES AND METHODS OF THE DECODER PLUG-IN

6.3.3 Custom Demuxer

The custom demuxer is implemented in the file “/vlcAstrals/modules/Demux/demuxdump.cpp”. The custom Demuxer access the buffer where the segments are, and starts to take set of pictures belonging to the same time instant (pictures from the multiple views). These are arranged as chunks. A chunk is supplied to the Decoder plug-in. Some parameters like the frame rate, video size and the Sequence Parameter Set (SPS) and the Picture Parameter Set (PPS) are needed by the decoder before it can decode pictures. This information is retrieved by the custom demuxer from an initialization segment, as suggested by (16), or the MPD. The chunk of the bit stream taken from the buffer

is contained in the “*block_t*” structure. This structure is passed to the decoder plug-in for decoding. This structure contains also timing fields. The “*i_dts*” field contains the time for decoding the current packet of bit stream. The “*i_pts*” field contains the time for presenting the decoded image to the display. [TABLE 9] lists some of the important structures and methods of the custom demuxer.

Structure	Description
Es_format_t fmt	Structure to save info needed by the decoder. (width, height, frame rate, sps, pps)
Block_t *p_data	Structure to save a set of pictures belonging to the same time instant. And some other properties like the pps and dts.
Method	Description
Es_out_add()	Sends to decoder the info contained in <i>fmt</i>
Stream_read()	Reads a set of pictures belonging to the same time instant from the buffer.
Es_out_Send()	Sends <i>p_data</i> to Decoder.

TABLE 9 IMPORTANT STRUCTURES AND METHODS OF THE CUSTOM DEMUXER

6.3.4 Custom HTTP Client

The custom HTTP client is the core of the adaptive HTTP streaming implementation at client side. It is implemented in the file “*/vlcAstrals/modules/Access/http.cpp*”. This custom module, besides provide typical HTTP progressive download capabilities, also provides:

1. Functions to recognize and analyze an arriving MPD in order to update services, playlists, and graphic user interface.
2. Efficient buffer management processes: These processes will try to keep constant the buffer size in terms of a predefined minimum number of buffered segments and will request new segments when necessary.
3. Bitrate estimation functions in order to apply proper adaptation procedures. These functions implement bandwidth fluctuation considerations, and are part of the optimized buffered scheme for adaptive HTTP streaming solutions. The details of such functions are not provided in this thesis work as they contain sensible information for STM, and innovative solutions proposed are potentially patent pending.
4. The custom http client module allows basic playback tricks functionality.

[TABLE 10] lists the higher level methods of the custom HTTP client.

Method	Description
Open() [Fig 46]	Opens the HTTP streaming client; it initiates all structures and buffer. This function is also launch each time the user use a playback trick functionality. E.g. require a particular segment for a particular representation. This is done to free the buffer of any remaining previous segment.
Connect() [Fig 46]	Opens TCP connection with server if there was not any.
Request() [Fig 47]	Constructs HTTP request for the selected segments or media.
MPD() [Fig 48]	Saves or updates MPD file, analyzes it and updates playlists, user interface, available services, etc.
Read() [Fig 49]	A complex function to download the segments and put them in the buffer. It makes bit rate estimation, buffer size control, and adaptation to proper representations, and trigger new segments requests.

TABLE 10 IMPORTANT METHODS OF THE CUSTOM HTTP CLIENT

Each one of these methods also implement functions that check for connection problems or memory problems, etc. The high level processes, important for the streaming solution, are shown in the following flow charts.

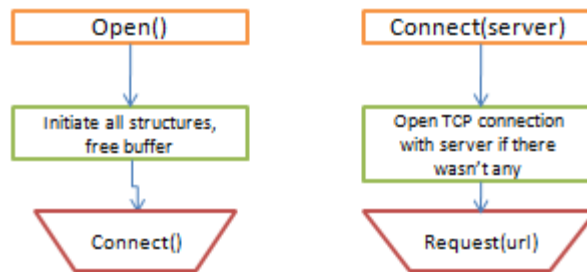


FIG 46 FLOW CHART FOR METHODS OPEN() AND CONNECT()

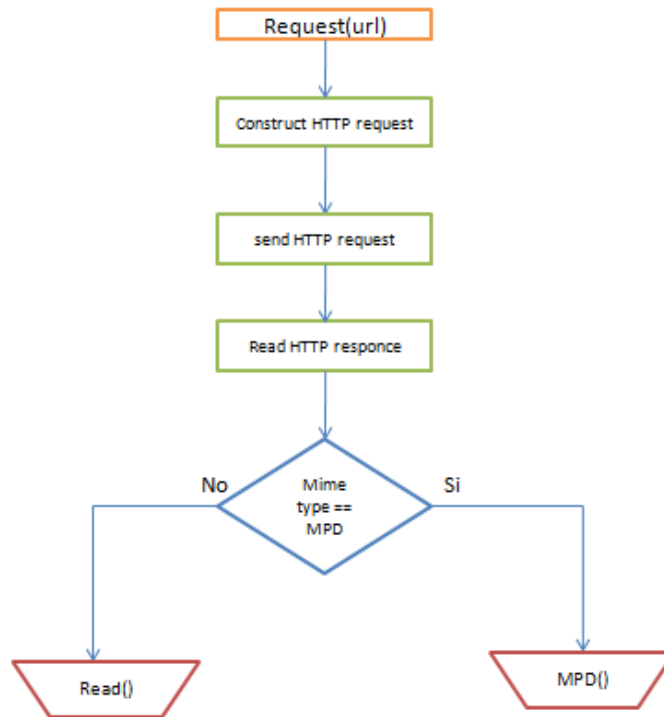


FIG 47 FLOW CHART FOR METHOD REQUEST()

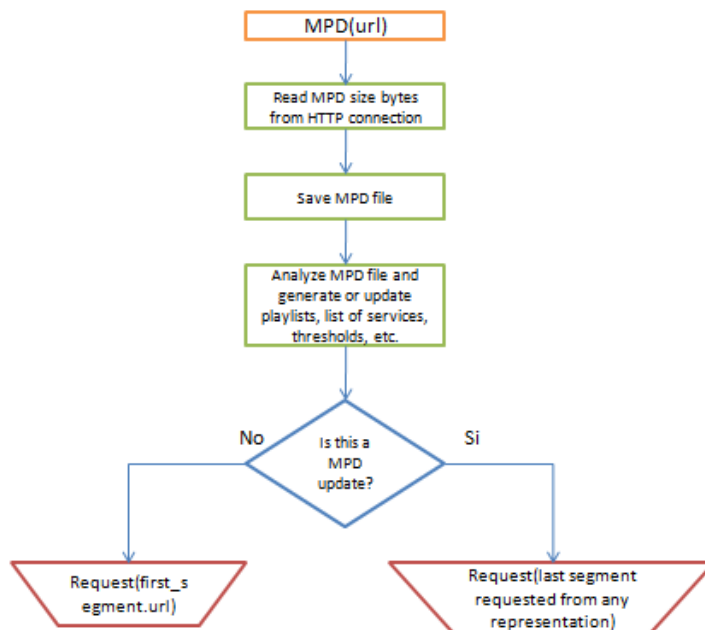


FIG 48 FLOW CHART FOR METHOD MPD()

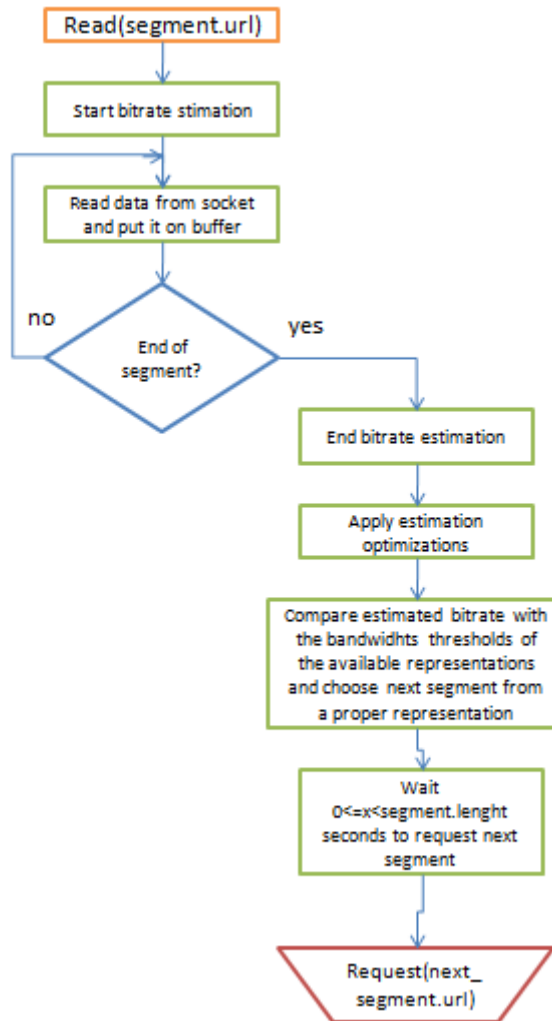


FIG 49 FLOW CHART FOR METHOD READ()

7

Test Bed, Tests and Results

A VoD test-bed was implemented. The test-bed consists of two adaptive HTTP streaming enabled servers and one adaptive HTTP streaming enabled client (see Fig 27 and Fig 50). The media resource for testing the framework is the two View-points video sequence Nurburgring race 3D, coded with H.264/MVC. The simulated scenarios cover examples of client automatic adaptation, adaptation at server side and traffic redistribution at CDN or server side in order enhance user experience and guarantee continuity of service. Bandwidth shapers are run on each server in order to trigger these adaptation and redistribution processes.

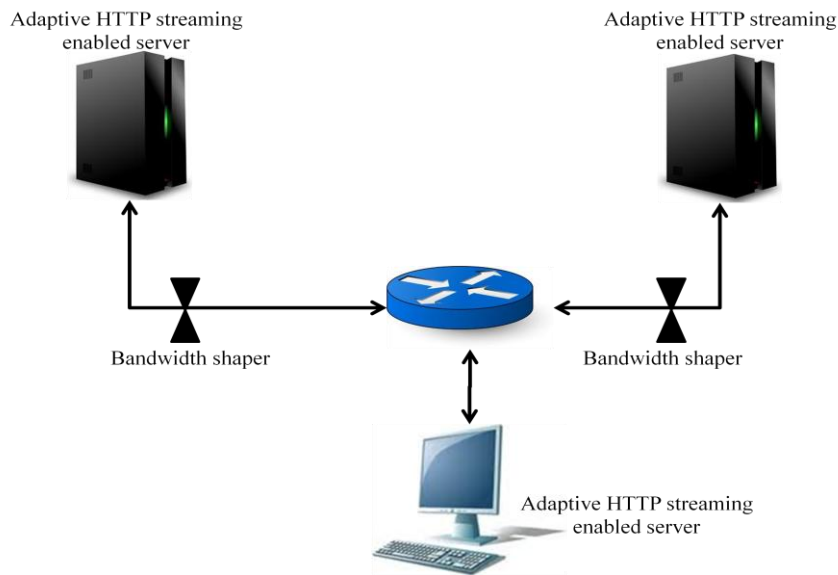


FIG 50 TEST BED ARCHITECTURE

7.1 Sequence for the test bed: Nurburgring race 3D

The media resource for testing the framework is the two View-points video sequence Nurburgring race. This video sequence has a frame rate of 30 FPS and a video size of 352x288 pixels. It was coded with H.264/MVC and fragmented in 11 segments, each one of 10 seconds of duration. The configuration file used for encoding each segment is shown in [TABLE 11]. The most important parameters from the framework point of view is the GOP size and Intra-Period. These parameters indicate that every 8 frames there is an anchor picture or intra-coded picture, which does not have any statistical dependencies on other frames, meaning this that they serve as Random Access Points (RAP) in case of seeking tricks using HTTP partial responses

```

#===== GENERAL =====
InputFile          nurburgring_segment_x      # input file
OutputFile         encoded_nurburgring_x      # bitstream file
SourceWidth        352                        # input frame width
SourceHeight       288                        # input frame height
FrameRate          30.0                       # frame rate [Hz]
FramesToBeEncoded  300                        # number of frames

#===== CODING =====
SymbolMode         1                         # 0=CAVLC, 1=CABAC
FRExt              1                         # 8x8 transform (0:off, 1:on)
BasisQP            31                         # Quantization parameters

#===== INTERLACED =====
MbAff              0                         # 0=frameMb, 1=MbAff
PAff               0                         # 0=frame, 1=field,
2=frame/field

#===== STRUCTURE =====
GOPSize            8                         # GOP Size (at maximum frame rate)
IntraPeriod        8                         # Anchor Period
NumberReferenceFrames 2                     # Number of reference pictures
InterPredPicsFirst 1                         # 1 Inter Pics; 0 Inter-view
Pics
DeltaLayer0Quant   0                         # differential QP for layer 0
DeltaLayer1Quant   3                         # differential QP for layer 1
DeltaLayer2Quant   4                         # differential QP for layer 2
DeltaLayer3Quant   5                         # differential QP for layer 3
DeltaLayer4Quant   6                         # differential QP for layer 4
DeltaLayer5Quant   7                         # differential QP for layer 5

#===== MOTION SEARCH =====
SearchMode         4                         # Search mode (0:BlockSearch, 4:FastSearch)
SearchFuncFullPel  3                         # Search function full pel
                                           SAD, 1:SSE, 2:HADAMARD, 3:SAD-YUV)
SearchFuncSubPel   2                         # Search function sub pel
                                           # (0:SAD, 1:SSE, 2:HADAMARD)
SearchRange        32                       # Search range (Full Pel)
BiPredIter         4                         # Max iterations for bi-pred search
IterSearchRange    8                         # Search range for iterations
                                           (0: normal)

#===== WEIGHTED PREDICTION =====
WeightedPrediction 0                         # Weighting IP Slice
(0:disable, 1:enable)
WeightedBiprediction 0                       # Weighting B Slice
                                           (0:disable, 1:explicit, 2:implicit)

#===== PARALLEL DECODING INFORMATION SEI Message =====
PDISEIMessage      0                         # PDI SEI message enable (0:
disable, 1:enable)
PDIInitialDelayAnc 2                         # PDI initial delay for anchor
pictures
PDIInitialDelayNonAnc 2                     # PDI initial delay for non-
anchor pictures

#===== VIEW SCALABILITY INFORMATION SEI MESSAGE =====

```

```

ViewScalInfoSEI          1          # (0: ViewScalSEI off, 1:
ViewScalSEI on)
#===== MULTIVIEW CODING PARAMETERS =====
NumViewsMinusOne         1          # (Number of view to be coded
minus 1)
ViewOrder                0-1      # (Order in which views are coded)
View_ID                  0          # view_id (0..1024): valid range
Fwd_NumAnchorRefs        0          # (number of list_0 references
for anchor)
Bwd_NumAnchorRefs        0          # (number of list 1 references for
anchor)
Fwd_NumNonAnchorRefs     0          # (number of list 0 references
for non-anchor)
Bwd_NumNonAnchorRefs     0          # (number of list 1 references
for non-anchor)
View_ID                  1          # view_id
Fwd_NumAnchorRefs        1          # (number of list_0 references
for anchor)
Bwd_NumAnchorRefs        0          # (number of list 1 references
for anchor)
Fwd_NumNonAnchorRefs     0          # (number of list 0 references
for non-anchor)
Bwd_NumNonAnchorRefs     0          # (number of list 1 references
for non-anchor)
Fwd_AnchorRefs           0 0        #ref_idx view_id combination
Bwd_AnchorRefs           0 2        #ref_idx view_id combination
Fwd_NonAnchorRefs        0 0        #ref_idx view_id combination

```

TABLE 11 ENCODER CONFIGURATION FILE FOR NURBURGRING SEQUENCE

The elementary stream of this coded sequence has two View-Points, which serve, for example, to offer a 3D service. From this elementary stream it is possible to extract or filter a lighter substream or representation containing just the base view. These 2 streams, the elementary and its substream, indicates server that two services could be offered to users: a 2-D service and a 3D service. Any of them can be hidden or offered to the client using different MPD profiles (see section 4.2.2). The 2-D representation needs a minimum required bandwidth of 350 Kbit/s and the 3D representation needs a minimum required bandwidth of 500 Kbit/s.

The available MPD profiles available at server side are explained in [TABLE 12].

MPD profile	Description
Main profile [Fig 51]	All services offered, 2-D and 3D representations
Traffic redistribution profile [Fig 52]	All services offered, 2-D and 3D representations, reallocate user from server 192.168.1.2 to 192.168.1.3..
Server adaptation profile [Fig 53]	Only 2-D representation available-

TABLE 12 MPD PROFILES FOR THE SEQUENCE NURBURGRING RACE

```

- <MPD type="OnDemand" minBufferTime="10S">
- <ProgramInformation moreInformationURL="http://www.st.com">
  <Title language="english">nurburgring @ 2 views</Title>
  <Source xml="jehg">STmicroelectronics</Source>
  <Copyright prop="prop">STmicroelectronics, All rights reserved</Copyright>
</ProgramInformation>
- <Period segmentAlignmentFlag="True">
- <Representation bandwidth="350" width="352" height="288" mimeType="video/mp4" startWithRAP="False">
- <SegmentInfo segments="10" duration="10">
  - <!--
    InitialisationSegmentURL sourceURL="FIFA-v96k-a48k_i.3gp"/
  -->
  <Url sourceURL="http://192.168.1.2/cgi-bin/nurburgring?segment=0_2D" name="segment_0_2D"/>
  <Url sourceURL="http://192.168.1.2/cgi-bin/nurburgring?segment=1_2D" name="segment_1_2D"/>
  <Url sourceURL="http://192.168.1.2/cgi-bin/nurburgring?segment=2_2D" name="segment_2_2D"/>
  <Url sourceURL="http://192.168.1.2/cgi-bin/nurburgring?segment=3_2D" name="segment_3_2D"/>
  <Url sourceURL="http://192.168.1.2/cgi-bin/nurburgring?segment=4_2D" name="segment_4_2D"/>
  <Url sourceURL="http://192.168.1.2/cgi-bin/nurburgring?segment=5_2D" name="segment_5_2D"/>
  <Url sourceURL="http://192.168.1.2/cgi-bin/nurburgring?segment=6_2D" name="segment_6_2D"/>
  <Url sourceURL="http://192.168.1.2/cgi-bin/nurburgring?segment=7_2D" name="segment_7_2D"/>
  <Url sourceURL="http://192.168.1.2/cgi-bin/nurburgring?segment=8_2D" name="segment_8_2D"/>
  <Url sourceURL="http://192.168.1.2/cgi-bin/nurburgring?segment=9_2D" name="segment_9_2D"/>
  <Url sourceURL="http://192.168.1.2/cgi-bin/nurburgring?segment=10_2D" name="segment_10_2D"/>
  </SegmentInfo>
</Representation>
- <Representation bandwidth="500" width="352" height="288" mimeType="video/mp4" startWithRAP="False">
- <SegmentInfo segments="10" duration="10">
  - <!--
    InitialisationSegmentURL sourceURL="FIFA-v96k-a48k_i.3gp"/
  -->
  <Url sourceURL="http://192.168.1.2/cgi-bin/nurburgring?segment=0_3D" name="segment_0_3D"/>
  <Url sourceURL="http://192.168.1.2/cgi-bin/nurburgring?segment=1_3D" name="segment_1_3D"/>
  <Url sourceURL="http://192.168.1.2/cgi-bin/nurburgring?segment=2_3D" name="segment_2_3D"/>
  <Url sourceURL="http://192.168.1.2/cgi-bin/nurburgring?segment=3_3D" name="segment_3_3D"/>
  <Url sourceURL="http://192.168.1.2/cgi-bin/nurburgring?segment=4_3D" name="segment_4_3D"/>
  <Url sourceURL="http://192.168.1.2/cgi-bin/nurburgring?segment=5_3D" name="segment_5_3D"/>
  <Url sourceURL="http://192.168.1.2/cgi-bin/nurburgring?segment=6_3D" name="segment_6_3D"/>
  <Url sourceURL="http://192.168.1.2/cgi-bin/nurburgring?segment=7_3D" name="segment_7_3D"/>
  <Url sourceURL="http://192.168.1.2/cgi-bin/nurburgring?segment=8_3D" name="segment_8_3D"/>
  <Url sourceURL="http://192.168.1.2/cgi-bin/nurburgring?segment=9_3D" name="segment_9_3D"/>
  <Url sourceURL="http://192.168.1.2/cgi-bin/nurburgring?segment=10_3D" name="segment_10_3D"/>
  </SegmentInfo>
</Representation>
</Period>
</MPD>

```

FIG 51 NURBURGRING RACE MAIN MPD PROFILE

```

- <Period segmentAlignmentFlag="True">
- <Representation bandwidth="208" width="352" height="288" mimeType="video/mp4" startWithRAP="False">
- <SegmentInfo segments="10" duration="10">
  <Url sourceURL="http://192.168.1.3/cgi-bin/nurburgring?segment=0_2D" name="segment_0_2D"/>
  <Url sourceURL="http://192.168.1.3/cgi-bin/nurburgring?segment=1_2D" name="segment_1_2D"/>
  <Url sourceURL="http://192.168.1.3/cgi-bin/nurburgring?segment=2_2D" name="segment_2_2D"/>
  <Url sourceURL="http://192.168.1.3/cgi-bin/nurburgring?segment=3_2D" name="segment_3_2D"/>
  <Url sourceURL="http://192.168.1.3/cgi-bin/nurburgring?segment=4_2D" name="segment_4_2D"/>
  <Url sourceURL="http://192.168.1.3/cgi-bin/nurburgring?segment=5_2D" name="segment_5_2D"/>
  <Url sourceURL="http://192.168.1.3/cgi-bin/nurburgring?segment=6_2D" name="segment_6_2D"/>
  <Url sourceURL="http://192.168.1.3/cgi-bin/nurburgring?segment=7_2D" name="segment_7_2D"/>
  <Url sourceURL="http://192.168.1.3/cgi-bin/nurburgring?segment=8_2D" name="segment_8_2D"/>
  <Url sourceURL="http://192.168.1.3/cgi-bin/nurburgring?segment=9_2D" name="segment_9_2D"/>
  <Url sourceURL="http://192.168.1.3/cgi-bin/nurburgring?segment=10_2D" name="segment_10_2D"/>
  </SegmentInfo>
</Representation>
- <Representation bandwidth="345" width="352" height="288" mimeType="video/mp4" startWithRAP="False">
- <SegmentInfo segments="10" duration="10">
  <Url sourceURL="http://192.168.1.3/cgi-bin/nurburgring?segment=0_3D" name="segment_0_3D"/>
  <Url sourceURL="http://192.168.1.3/cgi-bin/nurburgring?segment=1_3D" name="segment_1_3D"/>
  <Url sourceURL="http://192.168.1.3/cgi-bin/nurburgring?segment=2_3D" name="segment_2_3D"/>
  <Url sourceURL="http://192.168.1.3/cgi-bin/nurburgring?segment=3_3D" name="segment_3_3D"/>
  <Url sourceURL="http://192.168.1.3/cgi-bin/nurburgring?segment=4_3D" name="segment_4_3D"/>
  <Url sourceURL="http://192.168.1.3/cgi-bin/nurburgring?segment=5_3D" name="segment_5_3D"/>
  <Url sourceURL="http://192.168.1.3/cgi-bin/nurburgring?segment=6_3D" name="segment_6_3D"/>
  <Url sourceURL="http://192.168.1.3/cgi-bin/nurburgring?segment=7_3D" name="segment_7_3D"/>
  <Url sourceURL="http://192.168.1.3/cgi-bin/nurburgring?segment=8_3D" name="segment_8_3D"/>
  <Url sourceURL="http://192.168.1.3/cgi-bin/nurburgring?segment=9_3D" name="segment_9_3D"/>
  <Url sourceURL="http://192.168.1.3/cgi-bin/nurburgring?segment=10_3D" name="segment_10_3D"/>
  </SegmentInfo>
</Representation>
</Period>
</MPD>

```

FIG 52 NURBURGRING RACE TRAFFIC REDISTRIBUTION MPD PROFILE

```

- <MPD type="OnDemand" minBufferTime="10S">
- <ProgramInformation moreInformationURL="http://www.st.com">
  <Title language="english">nurburgring @ 2 views</Title>
  <Source xml="jehg">STmicroelectronics</Source>
  <Copyright prop="prop">STmicroelectronics, All rights reserved</Copyright>
</ProgramInformation>
- <Period segmentAlignmentFlag="True">
- <Representation bandwidth="208" width="352" height="288" mimeType="video/mp4" startWithRAP="False">
- <SegmentInfo segments="10" duration="10">
  <Url sourceURL="http://192.168.1.3/cgi-bin/nurburgring?segment=0_2D" name="segment_0_2D"/>
  <Url sourceURL="http://192.168.1.3/cgi-bin/nurburgring?segment=1_2D" name="segment_1_2D"/>
  <Url sourceURL="http://192.168.1.3/cgi-bin/nurburgring?segment=2_2D" name="segment_2_2D"/>
  <Url sourceURL="http://192.168.1.3/cgi-bin/nurburgring?segment=3_2D" name="segment_3_2D"/>
  <Url sourceURL="http://192.168.1.3/cgi-bin/nurburgring?segment=4_2D" name="segment_4_2D"/>
  <Url sourceURL="http://192.168.1.3/cgi-bin/nurburgring?segment=5_2D" name="segment_5_2D"/>
  <Url sourceURL="http://192.168.1.3/cgi-bin/nurburgring?segment=6_2D" name="segment_6_2D"/>
  <Url sourceURL="http://192.168.1.3/cgi-bin/nurburgring?segment=7_2D" name="segment_7_2D"/>
  <Url sourceURL="http://192.168.1.3/cgi-bin/nurburgring?segment=8_2D" name="segment_8_2D"/>
  <Url sourceURL="http://192.168.1.3/cgi-bin/nurburgring?segment=9_2D" name="segment_9_2D"/>
  <Url sourceURL="http://192.168.1.3/cgi-bin/nurburgring?segment=10_2D" name="segment_10_2D"/>
  </SegmentInfo>
</Representation>
</Period>
</MPD>

```

FIG 53 NURBURGRING RACE ADAPTATION MPD PROFILE

7.2 Setting up server

It is possible to start Null HTTP server with a call like the one illustrated in [CALL 6].

```
quacchio@quacchio:~/Tesi/nullhttpd-0.5.1/httpd/bin$ sudo ./httpd
sudo: unable to resolve host quacchio
Null httpd 0.5.1
Binding to 'http://any:80/'...OK.
```

CALL 6 LAUNCH NULL HTTP SERVER

Then launch the bandwidth shaper script with a call like the one illustrated in [CALL 7]

```
quacchio@quacchio:~/Tesi/nullhttpd-0.5.1/httpd/bin$ sudo ./tc.bash start
sudo: unable to resolve host quacchio
Starting bandwidth shaping: done
```

CALL 7 LAUNCH BANDWIDTH SHAPER

It is possible to edit some parameters of the bandwidth shaper by editing the shell file tc.bash. An example is shown in [Fig 54]

```
#!/bin/bash
#
# tc uses the following units when passed as a parameter.
# kbps: Kilobytes per second
# mbps: Megabytes per second
# kbit: Kilobits per second
# mbit: Megabits per second
# bps: Bytes per second
# Amounts of data can be specified in:
# kb or k: Kilobytes
# mb or m: Megabytes
# mbit: Megabits
# kbit: Kilobits
# To get the byte figure from bits, divide the number by 8 bit
#
TC=/sbin/tc
IF=eth0          # Interface
DNLD=1000kbit    # DOWNLOAD Limit
UPLD=1000kbit    # UPLOAD Limit
IP=192.168.1.2 | # Host IP
U32="$TC filter add dev $IF protocol ip parent 1:0 prio 1 u32"
```

FIG 54 EDITING BANDWIDTH SHAPER PARAMETERS

7.3 Requesting a media resource with VLC

[CALL 8] illustrate a media request using VLC, particularly, the Nurburgring media resource. It is possible to see in the URL that a MPD is being requested explicitly to the adaptive streaming application at server side.

```
>./vlc -codec h264mvc_decoder -sout-h264-mvc-force-video -sout-h264-mvc-flag3d 1
--sout-h264-mvc-viewpoint 0 -demux=dump
http://192.168.1.3/cgi-bin/nurburgring?segment=nurburging_MPD.3gm.
```

CALL 8 REQUEST THE NURBURGRING MEDIA RESOURCE WITH VLC

[TABLE 13] explain the parameters used in [CALL 8].

Parameter	Description
-codec h264mvc_decoder	Force VLC to use the H.264 MVC decoder
-sout-h264-mvc-force-video	Force VLC to display decoded frames even if they are late respect the presentation time.
-sout-h264-mvc-flag3d 1	If there are two view-points then use them to generate a 3D anaglyph composition of the video.
--sout-h264-mvc-viewpoint 0	Default view-point
-demux=dump	Force VLC to use custom demux.

TABLE 13 PARAMETERS PASSED TO VLC WITH THE LAUNCH CALL

In the next tests every streaming session initiate with a HTTP request for a MPD file, like the one illustrated in this section.

7.4 Test: Traffic Redistribution and Automatic Adaptation

The setup is composed by 2 servers and one client. In order to test switch from one overloaded server to another and to trigger automatic adaptation at client side, bandwidth shapers are run on each server. Server 1 and server 2 are identified by the IP addresses 192.168.1.3 and 192.168.1.2, and their available bandwidth has been set to 350Kbit/s and 1000 Kbit/s respectively [Fig 55].

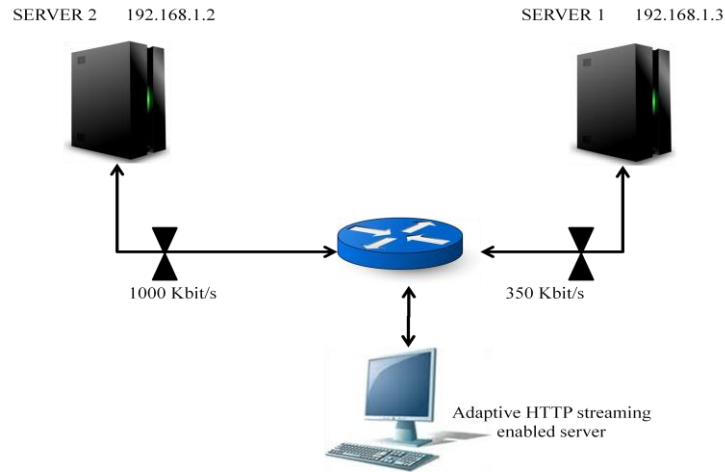


FIG 55 TEST BED FOR THE FIRST TEST

The experiment follows the flow chart in [Fig 56], which is generated with Wireshark (31), a network protocol analyzer for UNIX and Windows:

Time	192.168.1.1	192.168.1.2	192.168.1.3	Comment
13.074			GET /cgi-bin/nurbur	HTTP: GET /cgi-bin/nurburgring?segment=nurburgring_MPD.3gm HTTP/1.1
13.148			HTTP/1.1 200 OK	HTTP/XML: HTTP/1.1 200 OK
13.474			GET /cgi-bin/nurbur	HTTP: GET /cgi-bin/nurburgring?segment=0_2D HTTP/1.1
13.512			HTTP/1.1 200 OK	HTTP/XML: HTTP/1.1 200 OK
13.828			GET /cgi-bin/nurbur	HTTP: GET /cgi-bin/nurburgring?segment=1_2D HTTP/1.1
20.166			GET /cgi-bin/nurbur	HTTP: GET /cgi-bin/nurburgring?segment=2_2D HTTP/1.1
30.160			GET /cgi-bin/nurbur	HTTP: GET /cgi-bin/nurburgring?segment=3_2D HTTP/1.1
30.199			HTTP/1.1 200 OK	HTTP/XML: HTTP/1.1 200 OK
40.160			GET /cgi-bin/nurbur	HTTP: GET /cgi-bin/nurburgring?segment=4_2D HTTP/1.1
50.160			GET /cgi-bin/nurbur	HTTP: GET /cgi-bin/nurburgring?segment=5_3D HTTP/1.1
60.160			GET /cgi-bin/nurbur	HTTP: GET /cgi-bin/nurburgring?segment=6_3D HTTP/1.1
70.160			GET /cgi-bin/nurbur	HTTP: GET /cgi-bin/nurburgring?segment=7_3D HTTP/1.1
80.160			GET /cgi-bin/nurbur	HTTP: GET /cgi-bin/nurburgring?segment=8_3D HTTP/1.1
90.160			GET /cgi-bin/nurbur	HTTP: GET /cgi-bin/nurburgring?segment=9_3D HTTP/1.1

FIG 56 HTTP REQUESTS FLOW GRAPH

1. Client requests the video resource *Nurburgring race* to server 1.(Time 13.074). This is done with the call presented in [CALL 8]
2. Server 1 presents the media resource to the client sending the MPD file shown in [Fig 51] (time 13.148). The client analyzes the MPD file and constructs a playlist with all the segments from the 2D and 3D representation as illustrated in [Fig 57]

```
[00000236] main input debug: 'http://192.168.1.3/cgi-bin/nurburgring?segment=nurburgring_MPD.3gm' gives access 'http' demux 'nurburgring?segment=nurburgring_MPD.3gm'
[00000236] main input debug: Enforce demux 'dump'
[00000236] main input debug: creating access 'http' path='192.168.1.3/cgi-bin/nurburgring?segment=nurburgring_MPD.3gm'
[00000237] main access debug: looking for access2 module: 7 candidates
[00000237] access http access debug: http: server='192.168.1.3' port=80 file='/cgi-bin/nurburgring?segment=nurburgring_MPD.3gm'
[00000237] main access debug: net: connecting to 192.168.1.3 port 80
[00000237] main access debug: connection in progress
[00000237] access http access debug: protocol 'HTTP' answer code 200
[00000237] access http access debug: MPD size=3231
[00000231] main playlist debug: adding playlist item 'segment 0 2D' ( http://192.168.1.3/cgi-bin/nurburgring?segment=0_2D )
[00000231] main playlist debug: adding playlist item 'segment 1 2D' ( http://192.168.1.3/cgi-bin/nurburgring?segment=1_2D )
[00000231] main playlist debug: adding playlist item 'segment 2 2D' ( http://192.168.1.3/cgi-bin/nurburgring?segment=2_2D )
[00000231] main playlist debug: adding playlist item 'segment 3 2D' ( http://192.168.1.3/cgi-bin/nurburgring?segment=3_2D )
[00000231] main playlist debug: adding playlist item 'segment 4 2D' ( http://192.168.1.3/cgi-bin/nurburgring?segment=4_2D )
[00000231] main playlist debug: adding playlist item 'segment 5 2D' ( http://192.168.1.3/cgi-bin/nurburgring?segment=5_2D )
[00000231] main playlist debug: adding playlist item 'segment 6 2D' ( http://192.168.1.3/cgi-bin/nurburgring?segment=6_2D )
[00000231] main playlist debug: adding playlist item 'segment 7 2D' ( http://192.168.1.3/cgi-bin/nurburgring?segment=7_2D )
[00000231] main playlist debug: adding playlist item 'segment 8 2D' ( http://192.168.1.3/cgi-bin/nurburgring?segment=8_2D )
[00000231] main playlist debug: adding playlist item 'segment 9 2D' ( http://192.168.1.3/cgi-bin/nurburgring?segment=9_2D )
[00000231] main playlist debug: adding playlist item 'segment 10 2D' ( http://192.168.1.3/cgi-bin/nurburgring?segment=10_2D )
[00000231] main playlist debug: adding playlist item 'segment 0 3D' ( http://192.168.1.3/cgi-bin/nurburgring?segment=0_3D )
[00000231] main playlist debug: adding playlist item 'segment 1 3D' ( http://192.168.1.3/cgi-bin/nurburgring?segment=1_3D )
[00000231] main playlist debug: adding playlist item 'segment 2 3D' ( http://192.168.1.3/cgi-bin/nurburgring?segment=2_3D )
[00000231] main playlist debug: adding playlist item 'segment 3 3D' ( http://192.168.1.3/cgi-bin/nurburgring?segment=3_3D )
[00000231] main playlist debug: adding playlist item 'segment 4 3D' ( http://192.168.1.3/cgi-bin/nurburgring?segment=4_3D )
[00000231] main playlist debug: adding playlist item 'segment 5 3D' ( http://192.168.1.3/cgi-bin/nurburgring?segment=5_3D )
[00000231] main playlist debug: adding playlist item 'segment 6 3D' ( http://192.168.1.3/cgi-bin/nurburgring?segment=6_3D )
[00000231] main playlist debug: adding playlist item 'segment 7 3D' ( http://192.168.1.3/cgi-bin/nurburgring?segment=7_3D )
[00000231] main playlist debug: adding playlist item 'segment 8 3D' ( http://192.168.1.3/cgi-bin/nurburgring?segment=8_3D )
[00000231] main playlist debug: adding playlist item 'segment 9 3D' ( http://192.168.1.3/cgi-bin/nurburgring?segment=9_3D )
[00000231] main playlist debug: adding playlist item 'segment 10 3D' ( http://192.168.1.3/cgi-bin/nurburgring?segment=10_3D )
```

FIG 57 VLC ANALYZE MPD AND GENERATE PLAYLIST WITH 2D AND 3D SEGMENTS

- The client requests the first media segment, with the lowest bit rate representation (2-D representation) for a fast start-up. (Time 13.474). Screenshots from the log and VLC are shown in [Fig 58]

```
main private debug: received first data for our buffer
main input debug: creating demux: access='http' demux='dump' path='192.168.1.3/cgi-bin/nurburgring?segment=0_2D'
main demuxer debug: looking for demux2 module: 1 candidate
demuxdump demuxer: dumping raw stream to file 'stream-demux.dump'
main input debug: selecting program id=0
main module debug: using demux2 module "demuxdump"
main decoder debug: looking for decoder module: 18 candidates
h264mvc_decoder decoder debug: STMicroelectronics H.264/MVC Optimized decoder
h264mvc_decoder decoder debug: *** User options: error resilience: 0, max POC diff: 1000, force display: TRUE
h264mvc_decoder decoder debug: *** Read a MP4 Configuration box containing 52 bytes
h264mvc_decoder decoder debug: *** Found a MPEG-4 FileFormat compliant bitstream
main module debug: using decoder module "h264mvc_decoder"
main decoder debug: thread 3938440976 (decoder) created at priority 0 (src/input/decoder.c:159)
main input debug: 'http://192.168.1.3/cgi-bin/nurburgring?segment=0_2D' successfully opened
```

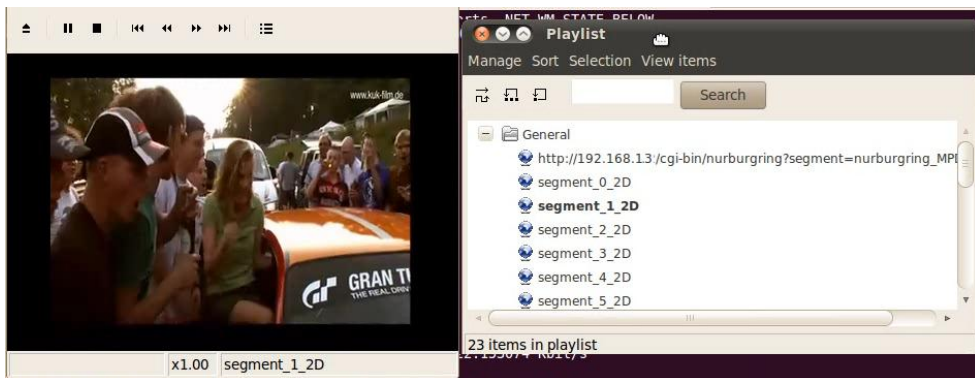


FIG 58 VLC REQUESTS THE FIRST MEDIA SEGMENT (2-D REPRESENTATION)

- The client estimates the available bandwidth and decides to stay on the 2D representation because server1 doesn't offer enough bandwidth. (Time 13.828 to 30.160)s.

5. A control Mechanism at CDN side monitoring servers load, instructs server1 to send an updated MPD to the connected client. The new MPD will contain URLs pointing to the offloaded server2 (Time 30.199)
6. The client updates the MPD file and starts requesting segments to server 2; segments still belong to the 2D representation. Seamless switching between servers is completed. (Time 40.160)

```
segment: 4 size 197271 bytes in 1.656911 seconds at 930.152366 Kbit/s

[00000250] main decoder debug: thread 3938440976: secret message triggered
[00000247] access_http access debug:

    est_Bitrate = 664.000000 threshold =500
[00000247] main access debug: net: connecting to 192.168.1.2 port 80
[00000247] main access debug: connection in progress
[00000247] access_http access debug: protocol 'HTTP' answer code 200
[00000247] access_http access debug:
segment: 5 size 223002 bytes in 1.912203 seconds at 911.097370 Kbit/s

[00000247] access_http access debug:

    est_Bitrate = 854.000000 threshold =500
[00000247] main access debug: net: connecting to 192.168.1.2 port 80
[00000247] main access debug: connection in progress
[00000247] access_http access debug: protocol 'HTTP' answer code 200
[00000247] access_http access debug:
segment: 6 size 377388 bytes in 3.152664 seconds at 935.191238 Kbit/s

[00000247] access_http access debug:

    est_Bitrate = 925.000000 threshold =500
```

FIG 59 SERVER 2 HAS MORE BANDWIDTH

7. The client continues to estimate the available bandwidth and decides to switch to the 3D representation. (Time 50.150 to 90.160)



FIG 60 VLC DISPLAYING A 3D REPRESENTATION

[Fig 61] illustrates the traffic flow between server and client. The black spikes represent HTTP requests, red and green graphs represent the TCP packets flow carrying segments between the client and servers 1 and 2 respectively. The number over the graphs represents the transmitted segment. It is possible to see how server 2 is faster than server 1 and how client waits some time before request the next segment. The graphic is done using Wireshark suite.

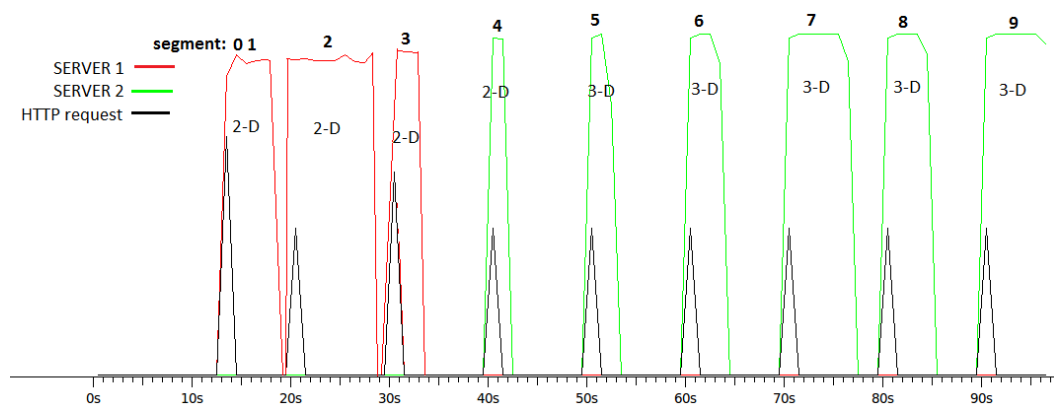


FIG 61 TRAFFIC REALLOCATION TO ENHANCE QOS

[Fig 62] shows the bitrate estimated by the client after downloading each segment, and how it is compared with a threshold (average bandwidth for 3D viewpoint) contained in the MPD file as a criterion for switching to the 3D representation. It is possible to see that the switch doesn't come instantly after changing media server, because this process is transparent to the client. The client change to the new representation once the estimated bitrate for a segment points out the possibility of switching to a better representation.

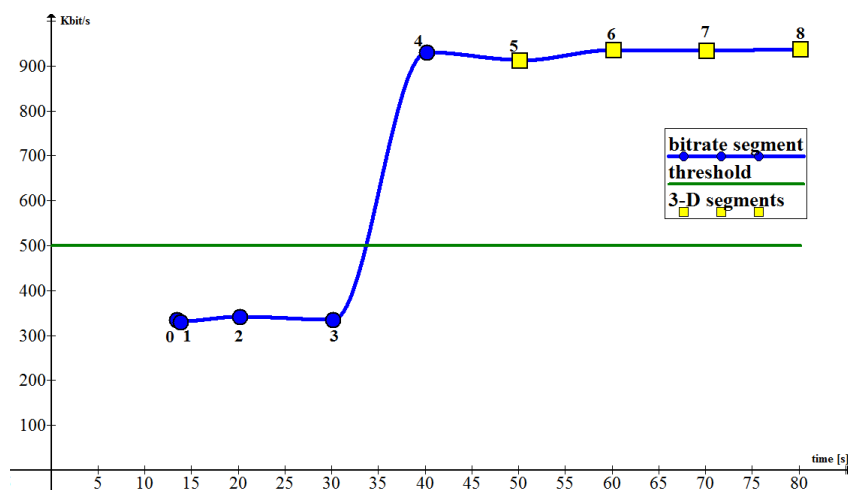


FIG 62 CLIENT'S ADAPTATION PROCESS AFTER A TRAFFIC REALLOCATION AT CDN SIDE

7.5 Test: Playback tricks and preferred 2-D/3D view points

The setup is composed by one server and one client. In this arrangement the client control mechanism has been set to let the buffer have 2 segments instead of one. This means that each time the user seek or request a different segment or representation, the client request at best effort the first three segments.

This test aims to show seeking tricks functionality and automatic adaptation at client side. The experiment follows the flow chart in [Fig 63].

Time .	Source	Destination	Protocol	Info
0.029672	192.168.1.1	192.168.1.2	HTTP	GET /cgi-bin/nurburgring?segment=nurburgring.MPD.3gm HTTP/1.1
0.055886	192.168.1.2	192.168.1.1	HTTP/XML	HTTP/1.1 200 OK
0.393237	192.168.1.1	192.168.1.2	HTTP	GET /cgi-bin/nurburgring?segment=0 2D HTTP/1.1
0.427599	192.168.1.2	192.168.1.1	HTTP/XML	HTTP/1.1 200 OK
0.745931	192.168.1.1	192.168.1.2	HTTP	GET /cgi-bin/nurburgring?segment=1 2D HTTP/1.1
2.976472	192.168.1.1	192.168.1.2	HTTP	GET /cgi-bin/nurburgring?segment=2 2D HTTP/1.1
6.432410	192.168.1.1	192.168.1.2	HTTP	GET /cgi-bin/nurburgring?segment=3 2D HTTP/1.1
16.432763	192.168.1.1	192.168.1.2	HTTP	GET /cgi-bin/nurburgring?segment=4 3D HTTP/1.1
26.433113	192.168.1.1	192.168.1.2	HTTP	GET /cgi-bin/nurburgring?segment=5 3D HTTP/1.1
36.433561	192.168.1.1	192.168.1.2	HTTP	GET /cgi-bin/nurburgring?segment=6 3D HTTP/1.1
46.433864	192.168.1.1	192.168.1.2	HTTP	GET /cgi-bin/nurburgring?segment=7 3D HTTP/1.1
47.014563	192.168.1.1	192.168.1.2	HTTP	GET /cgi-bin/nurburgring?segment=5 2D HTTP/1.1
48.117522	192.168.1.1	192.168.1.2	HTTP	GET /cgi-bin/nurburgring?segment=6 2D HTTP/1.1
49.953208	192.168.1.1	192.168.1.2	HTTP	GET /cgi-bin/nurburgring?segment=7 2D HTTP/1.1
60.129626	192.168.1.1	192.168.1.2	HTTP	GET /cgi-bin/nurburgring?segment=6 2D HTTP/1.1
61.964805	192.168.1.1	192.168.1.2	HTTP	GET /cgi-bin/nurburgring?segment=7 2D HTTP/1.1
65.426274	192.168.1.1	192.168.1.2	HTTP	GET /cgi-bin/nurburgring?segment=8 2D HTTP/1.1
75.596025	192.168.1.1	192.168.1.2	HTTP	GET /cgi-bin/nurburgring?segment=3 3D HTTP/1.1
78.515689	192.168.1.1	192.168.1.2	HTTP	GET /cgi-bin/nurburgring?segment=4 3D HTTP/1.1
80.856936	192.168.1.1	192.168.1.2	HTTP	GET /cgi-bin/nurburgring?segment=5 3D HTTP/1.1
90.852759	192.168.1.1	192.168.1.2	HTTP	GET /cgi-bin/nurburgring?segment=6 3D HTTP/1.1
100.853116	192.168.1.1	192.168.1.2	HTTP	GET /cgi-bin/nurburgring?segment=7 3D HTTP/1.1
111.046282	192.168.1.1	192.168.1.2	HTTP	GET /cgi-bin/nurburgring?segment=6 2D HTTP/1.1
112.883146	192.168.1.1	192.168.1.2	HTTP	GET /cgi-bin/nurburgring?segment=7 2D HTTP/1.1
116.348102	192.168.1.1	192.168.1.2	HTTP	GET /cgi-bin/nurburgring?segment=8 2D HTTP/1.1

FIG 63 HTTP REQUESTS FLOW GRAPH TEST 2

[Fig 64] illustrates the traffic flow between server and client. The black spikes represent HTTP requests; red graphs represent the TCP packets flow carrying segments between client and servers 1 and 2 respectively. The seeking tricks were requested in 47.01 s, 75.5s and 111.04 seconds, were a quick burst of HTTP requests can be seen, this behavior is explained by the buffer policies of try to buffer 2 segments as soon as possible. Other representations changes were made automatically by client as adaptation processes.

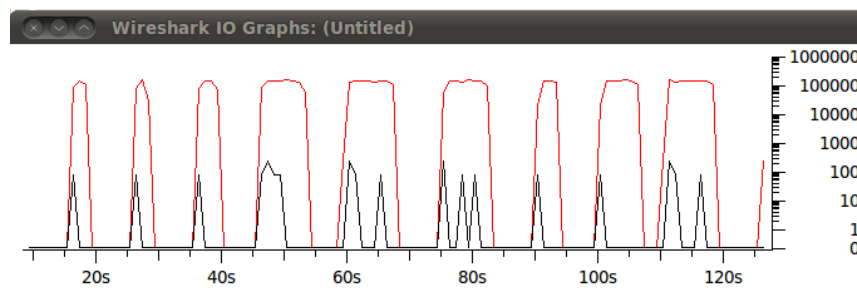


FIG 64 TCP FLOW BETWEEN SERVER AND CLIENT. TEST2

8

Future Work

In this thesis a framework for adaptive streaming of MultiView contents was presented. The work follows the guidelines issued by 3GPP that have been extended to support delivery and adaptation of Stereo and MultiView contents. The implemented framework enable smooth and quality streaming services over open internet for bandwidth demanding contents such as 3D video. On the other hand the solution proposed offers to services providers and CDN flexibility in managing and distributing contents.

This thesis work proposes some activities as future work. The framework could be extended in order to support advanced FreeView point streaming services and broadcast scenarios. Another possible activity is the development of algorithms to efficiently redistribute traffic at CDN side using the methods explained in this thesis.

8.1 Free View-Point streaming

Free Viewpoint Television allows the viewer to interactively choose his viewpoint within a 3D space in order to observe a dynamic scene from arbitrary perspectives, it must be remarked that 3DTV and FTV do not exclude each other; they should be combined in a single system to keep the advantages of both of them. In Multi-view TV, users can get only a limited view of a real 3D world they want to see and the view is determined not by users but by an array of cameras placed in the 3D world. FTV provides the new function to view the 3D world freely as if one is there. The function of view generation is thought as divided into two parts:

1. **Depth search:** all the information related to 3D space is used to generate a depth-map of the scene.
2. **Interpolation:** the depth-maps are exploited to synthesize new views.

Taking the above points into consideration, an FTV system can be designed in various ways as depicted in [Fig 65].

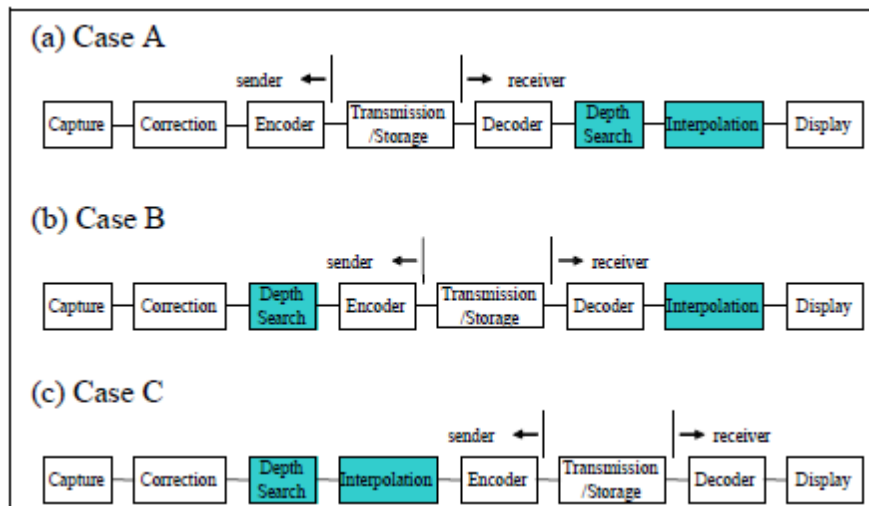


FIG 65 VARIOUS FTV SYSTEM ARCHITECTURES

In case A, both depth search and interpolation are performed at the receiver side. In case B, the depth search is performed at the sender side and the interpolation is performed at the receiver side. In case C, both depth search and interpolation are performed at the sender side. For example, the deep maps could be generated and sent by the sender or server using another HTTP session as depicted in [Fig 66].

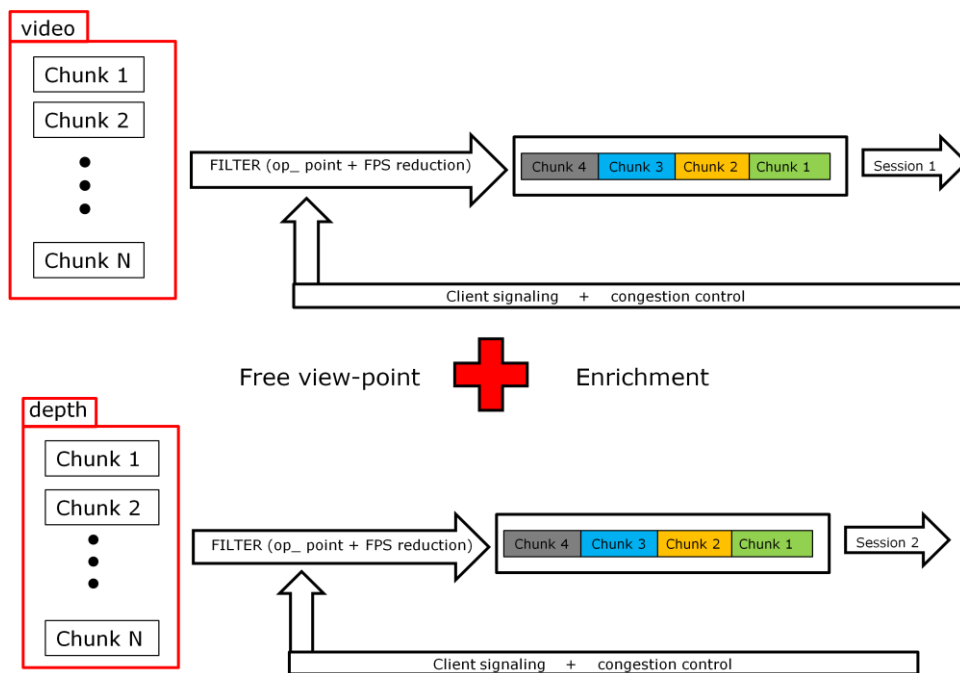


FIG 66 FVV STREAMING

8.2 Broadcast scenario

We explained broadcast scenarios in section 3.1.2. The extension of this framework to broadcast scenarios should consider that HTTP is typically used for distributed information systems, where performance can be improved by the use of response caches. There are a wide variety of architectures and configurations of caches and proxies currently being experimented with or deployed across the World Wide Web. These systems include national hierarchies of proxy caches to save transoceanic bandwidth, systems that broadcast or multicast cache entries and so on. The problem is that not all responses are usefully cacheable, and some requests may contain modifiers which place special requirements on cache behavior. Caching would be useless if it did not significantly improve performance. The goal of caching in HTTP/1.1 is to eliminate the need to send requests in many cases, and to eliminate the need to send full responses in many other cases. It uses "expiration" mechanisms and other more complex mechanisms for this purpose. Innovative systems to cache media segments could be used to efficiently enhance media delivery. For example in a live streaming session a cache system could only save the last two transmitted segments in a certain number of representations and transform any segment request into a request to one of those segments.

Bibliography

1. **Internet World Stats.** *Stats. World Internet Usage Statistics News and World Population Stats.* [Online] Miniwatts MArketing Group, June 31, 2010. <http://www.internetworldstats.com>.
2. **Morgan Stanley.** *Internet Trends.* 2010.
3. **VideoLan project.** VideoLAN -VLC: free streaming and multimedia solutions for all OS! [Online] VideoLAN. <http://www.videolan.org>.
4. **IBM.** IBM - Cloud computing. *IBM.* [Online] IBM, 2010. <http://www.ibm.com/ibm/cloud/>.
5. *Market-Oriented Cloud Computing: Vision, Hype, and Reality for Delivering IT Services as Computing Utilities.* **Buyya, Rajkumar and Chee Shin Yeo, Srikumar Venugopal.** s.l. : Department of Computer Science and Software Engineering, University of Melbourne, Australia, 2009.
6. *A vision for the Internet.* **Kleinrock, L.** s.l. : ST Journal of research, 2005.
7. **Zambelli, Alex.** *IIS Smooth Streaming Technical Overview.* s.l. : Microsoft Corporation, 2009.
8. *INTERACTIVE MULTI-VIEW VIDEO ADAPTATION FOR 3DTV.* **Ilkwon Park, Manbae Kim, Hong Kook Kim , and Hyeran Byun.** s.l. : 3DTV Conference: The True Vision - Capture, Transmission and Display of 3D Video, 2008.
9. **ITU-T.** *H.264 Advanced video coding for generic audiovisual services.* 2009.
10. *Efficient Prediction Structures for Multi-view Video Coding.* **Philipp Merkle, Aljoscha Smolic, Karsten Müller, and Thomas Wiegand.** 11, Berlin, Germany : IEEE Transactions on Circuits and Systems for Video Technology, 2007, Vol. 17. 1051-8215.
11. *CHALLENGES AND OPPORTUNITIES IN VIDEO CODING FOR 3D TV.* **Hari Kalva, Lakis Christodoulou, Liam Mayron, Oge Marques, and Borko Furht.** Dept. of Computer Science and Engineering, Florida Atlantic University, Boca Raton, FL : IEEE International Conference on Multimedia and Expo, 2006.

12. *The Emerging MVC Standard for 3D Video Services*. **Ying Chen, Ye-Kui Wang, Kemal Ugur, Miska M. Hannuksela, Jani Lainema and Moncef Gabbouj**. s.l. : EURASIP Journal on Advances in Signal Processing, 2009.
13. **Streaming Media, an Information Today Inc.** What is streaming? *streamingmedia*. [Online] <http://www.streamingmedia.com>.
14. **Microsoft**. Differences Between Multicast and Unicast. *Microsoft*. [Online] 2003. [Cited: september 27, 2010.] <http://support.microsoft.com/kb/291786>.
15. **Anand, A.** *Study and Implementation of Protocols for Transmission of H.264-MVC Coded Media Over Real-Time Packet Networks*. Turin, Italy : Thesis submitted to Politecnico di Torino, 2009.
16. **3GPP**. *3GPP TS 26.234 (Release 6) working draft on Transparent end-to-end Packet-switched Streaming service (PSS)*. s.l. : 3GPP, 2010.
17. **Wikipedia EN**. Data buffer (computer science). [Online] Wikipedia. [Cited: September 3, 2010.] [http://en.wikipedia.org/wiki/buffer_\(computer_science\)](http://en.wikipedia.org/wiki/buffer_(computer_science)).
18. *Adaptive playout strategies for packet video receivers with finite buffer capacity*. **Laoutaris, N. Stavrakakis, I.** s.l. : IEEE International Conference on Communications, 2001.
19. **John G. Apostolopoulos, Wai-tian Tan, Susie J. Wee**. *Video Streaming: Concepts, Algorithms, and Systems*. s.l. : Mobile and Media Systems Laboratory, HP Laboratories Palo Alto., 2002.
20. **Krasic, C. and Li, K. and Walpole, J.** The case for streaming multimedia with TCP. *Lecture Notes in Computer Science*. s.l. : Springer, 2001.
21. **Schulzrinne, H., Rao, A., and Lanphier, N.** "Real Time Streaming Protocol (RTSP)", *RFC 2326*. 1998.
22. **Handley, Mark and Van Jacobson, Colin Perkins**. "SDP: Session Description Protocol (RFC 4566)". 2008.
23. **James Wright, MSc**. *Session Description Protocol*. s.l. : Konnetic limited company.
24. **Fielding, Roy T., et al.** *RFC 2616: Hypertext Transfer Protocol -- HTTP/1.1*. s.l. : World Wide Web consortium, 1999.

25. **Colburn, Mike.** The Difference Between HTTP Progressive Download and True Streaming. *EzineArticles.com*. [Online] EzineArticles, 2010. [Cited: October 2, 2010.] <http://ezinearticles.com/?The-Difference-Between-HTTP-Progressive-Download-and-True-Streaming&id=2419776>.
26. **P2Pnext project.** *p2pnext deliverable. overview adaptive streaming systems*. s.l. : p2pnext, 2010.
27. **project of the Joint Video Team (JVT) of the ISO/IEC Moving Pictures Experts Group (MPEG) and the ITU-T Video Coding Experts Group (VCEG).** *MVC software manual, version: JMVC 8.0 (CVS tag: JMVC_8_0)*. 2010.
28. **NullLogic.** NullLogic httpd. [Online] <http://nullhttpd.sourceforge.net/httpd/>.
29. **Gundavaram, Shishir.** *CGI Programming on the World Wide Web*. s.l. : O'Reilly Media, 1st Edition March 1996.
30. “*Streaming content wars: Download and play strikes back*”. **Ch. Z. Patrikakis, N. Papaoulakis, Ch. Stefanoudaki, M. S. Nunes.** Venice, Italy : presented at the Personalization in Media Delivery Platforms Workshop , 2009.

Appendix A: Troubleshooting for the client's compilation process

Problem: JMVC does not compile.

Solution: JMVC does not compile with gcc versions newer than gcc 4.1. Download from gcc repository version 4.1, install it and prioritize it the PATH environmental variable of Linux, E.g.:

```
PATH=/usr/bin/GCC4.1:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
```

Problem: Compiling JMVC wrapping

Solution: Under 64 bits architectures force the flag -m64 in makefile.base /jmvc/JSVM/H264Extension/build/linux/common/makefile.base

Problem: Compiling VLC

Solution: download: libmpeg2, wx , libxml2, x11, x11vnc libraries. Then try to disable libraries unused in the HTTP streaming experiments.

```
./configure: disable-mkv --disable-realrtsp --disable-ffmpeg --enable-h264mvc --with-h264mvc=/home/STMH264MVCcodec/binaries/ --disable-STMLibrary -enable-release --enable-optimizations --disable-livedotcom --disable-mad --disable-libmpeg2 --disable-wxwidgets --disable-skins2.
```