

System Components for Video on Demand:  
Architecture and Implementation

-Doctoral Thesis-

Florin Lohan

7th May 2004

## Abstract

One of the main application of broadband multimedia is Video on Demand (VoD). This application allows its users to select and watch content (movies) in their homes, at any time they desire. There are still open issues about VoD that make this kind of service a reality of the future rather than a reality of the present.

The general scope of this thesis is to address some of these open issues and make some steps towards bringing high-quality, VoD multimedia content to people's TVs.

The thesis focuses on solving the following issues related with the VoD systems: insufficient interoperability between VoD components (client and servers), the high price and complexity of a VoD system and its components, lack of multimedia players on certain platforms, insufficient management of VoD systems and components, and the customers interface to the VoD system.

This thesis gives solutions for the above mentioned issues by proposing new architectures at several layers of the VoD system. The thesis also describes how to implement efficiently the proposed architectures. The main purpose of the presented implementation work is the validation of the presented architectures. The proposed solutions and architectures are software based and concern the application layer of the VoD system, its components, subcomponents, and communication protocols. Issues and solutions regarding the hardware and the lower layer protocols (e.g. physical, link, network or transport layers) are outside the scope of this thesis.

After the introductory chapter, which underlines the motivation behind this work, the thesis proposes solutions to the open issues mentioned above. Each proposed solution is described in a separate chapter. Chapter 2 introduces a package handling several multimedia communication protocols (RTSP, RTP and SDP) in an unified way, in order to lower the complexity of the remote content retrieval. Chapter 3 proposes a procedure for mapping DMIF primitives onto IETF protocols (RTSP, RTP and SDP) in order to create interoperability between VoD components which use DMIF, on one hand and VoD components which use the above-mentioned IETF protocols, on the other hand. Chapter 4 introduces a Linux-based networked multimedia client for playback of VoD content. The player is meant to run in both PC and STB environments, so that it can use either hardware-based or software-based media decoding. Chapter 5 presents a low-complexity method for determining the stream retrieval capability of the storage subsystem in a VoD server. This method is validated experimentally and it is shown that it is possible to achieve the high throughput necessary in a VoD server with cost-effective PC hardware. Chapter 6 introduces a web-based management platform that configures, monitors and controls several types of servers in a multimedia system that includes VoD services. This web-based management platform also provides an interface to the multimedia system for the customers. The web-based management platform gives a solution to both insufficient management of a VoD system and the customers interface to the VoD system. Chapter 7 presents the general conclusions, future challenges and open issues.

# Contents

<b>1</b>	<b>Introduction</b>	<b>13</b>
1.1	Problems in current VoD systems . . . . .	13
1.2	Problems outside the scope of this thesis . . . . .	15
1.3	Solutions proposed in this thesis . . . . .	17
1.4	Publications on which the thesis is based . . . . .	19
1.5	Outline of the thesis and main results . . . . .	20
1.6	Overview of existing solutions to the problems discussed in this thesis	22
<b>2</b>	<b>Integration of Communication Protocols</b>	<b>26</b>
2.1	Short presentation of RTSP, SDP and RTP protocols . . . . .	28
2.2	The architecture of the integrated RTSP, RTP and SDP library . . . .	30
2.2.1	The messages handling part . . . . .	32
2.2.2	The RTSP session handling part . . . . .	32
2.2.3	Implementation and performance issues . . . . .	36
2.3	Applications of the library . . . . .	37
2.3.1	Complete system for the transport of MPEG-4 content over the Internet . . . . .	37
2.3.2	Plugin-based networked multimedia player for PC and STB . . .	37
2.3.3	Video on Demand (VoD) multimedia server . . . . .	38
2.4	Summary and author's contributions . . . . .	38
<b>3</b>	<b>Transport of MPEG-4 over IP</b>	<b>40</b>
3.1	Delivery in MPEG-4 standard . . . . .	40
3.2	Related work . . . . .	42
3.3	Short presentation of DMIF and DAI . . . . .	44
3.4	Mapping of DAI primitives onto RTSP methods . . . . .	47
3.4.1	Important issues concerning mapping DMIF to RTSP . . . . .	48
3.4.2	Mapping of DMIF commands to RTSP, on the client side . . . .	52
3.4.3	Mapping of RTSP commands to DMIF, on the server side . . . .	61
3.4.4	Usage of the DAI_ChannelDescriptor QoS parameters when using RTSP as control protocol . . . . .	66
3.4.5	Mapping between DMIF response and RTSP response . . . . .	66
3.4.6	Interoperability and optimization issues for DMIF clients and servers which use RTSP for signalling . . . . .	66

3.4.7	A previous version of mapping of DMIF primitives to RTSP methods . . . . .	70
3.5	Implementation . . . . .	71
3.5.1	Performance evaluation . . . . .	73
3.6	Summary and author's contributions . . . . .	76
<b>4</b>	<b>The Architecture of a Multimedia Player</b>	<b>77</b>
4.1	Motivation . . . . .	78
4.2	History and analysis of Linux video players . . . . .	78
4.3	Related scientific work . . . . .	80
4.4	The player architecture . . . . .	81
4.4.1	General components description . . . . .	81
4.4.2	Plugins . . . . .	84
4.4.3	Tools . . . . .	85
4.4.4	Media playback with software decoding . . . . .	86
4.4.5	Media playback with hardware-aided decoding . . . . .	87
4.5	Summary and author's contributions . . . . .	88
<b>5</b>	<b>The Architecture of a VoD Server</b>	<b>89</b>
5.1	Introduction . . . . .	89
5.2	Designing VoD servers . . . . .	90
5.2.1	Application requirements: . . . . .	90
5.2.2	Cost requirements . . . . .	90
5.2.3	Deployment requirements . . . . .	91
5.2.4	Delivery in VoD systems . . . . .	91
5.2.5	Disk retrieval in VoD servers . . . . .	93
5.3	Behavioural model of the storage subsystem . . . . .	95
5.3.1	The Disk Pump algorithm . . . . .	96
5.3.2	The behavioural model algorithm . . . . .	96
5.4	Performance evaluation of the Disk Pump algorithm . . . . .	98
5.4.1	Test program and system . . . . .	98
5.4.2	Test results . . . . .	99
5.5	The architecture of a VoD server . . . . .	103
5.6	Summary and author's contributions . . . . .	104
<b>6</b>	<b>Management Platform for Multimedia System</b>	<b>105</b>
6.1	Related work . . . . .	106
6.2	System components and initial requirements . . . . .	107
6.2.1	VoD servers . . . . .	108
6.2.2	IP TV servers . . . . .	108
6.2.3	Transcoding servers . . . . .	109
6.2.4	Requirements and motivation . . . . .	110
6.3	System architecture . . . . .	111
6.3.1	The Server Application . . . . .	112
6.3.2	The Monitoring Daemon . . . . .	113

6.3.3	The Servers . . . . .	113
6.3.4	The User Gateway . . . . .	114
6.4	Implementation details . . . . .	114
6.5	Summary and author's contributions . . . . .	115
<b>7</b>	<b>Conclusions</b>	<b>117</b>

# Preface

"You have to find something that you love enough to be able to take risks, jump over the hurdles and break through the brick walls that are always going to be placed in front of you. If you don't have that kind of feeling for what it is you're doing, you'll stop at the first giant hurdle." - George Lucas -

The work for this thesis has been carried out during the years 1999-2003 at the Digital Media Institute, of Tampere University of Technology, Finland as part of several research projects.

First and foremost I wish to express my gratitude to my supervisor, Professor Irek Defée, for his guidance, constant support and patience during all these years, for the fact that he was always ready to talk with me, his door was never closed.

I would like to thank Professor Timo Hämäläinen from the Department of Mathematical Information Technology, University of Jyväskylä and Professor Andrew Perkis from the Department of Telecommunications, Norwegian University of Science and Technology, the reviewers of this thesis, for their constructive feedback and helpful comments.

For four years I had the privilege of being a postgraduate student in the Graduate School in Electronics, Telecommunications, and Automation (GETA), which partially funded my studies and which is gratefully acknowledged. I also wish to express my gratitude to Professor Iiro Hartimo, the Director of GETA, and to Marja Leppäharju, secretary-coordinator of GETA, for organizing enjoyable and interesting courses and seminars, and for their encouragements and recommendations during the annual GETA meetings.

This thesis would have looked different if I had not met Roberto Castagno. His advises and his attitude towards work and life formed me as a researcher and some of his remarks guided me through the later years of my thesis. My deepest gratitude for all!

Over the years I have had the privilege to work with a wonderful group of people. The sum of our achievements together is much more than any individual achievement, and together we have really built something. Aurelian Pop, Prakash Sastry, Mikko Suniala, Marius Vlad, friends and colleagues, I thank you from all my heart.

Warm thanks go also to my co-authors, Mika Rustari, Jami Kangasoja, Serkan Kiranyaz, Harri Hakulinen, Andrea Basso, Socrates Varakliotis, for very interesting talks together and fruitful cooperation.

I would also wish to thank Artur Lugmayr for the fun during our lengthy talks about what is good (C) and what is bad (Java) for our research, especially digital and interactive TV. Many thanks to Jens Spieker for the talks and for showing me other sides of research. Also many thanks to Pekka Koskinen, Riku Järvensivu, Corneliu Rusu, Amelia Gutierrez, Juan-Luis Medina, Santiago Gonzales, Gonzalo Valencia, Anssi Huttunen, Antti Mattila, office mates, colleagues and friends, for enjoyable time spent together. Warm thanks also to Florina, Corina and Victor, for making the work more enjoyable on so many occasions.

Warm thanks are due to Virve Larmila, Maarit Ahonen-Lehmus, Elina Orava and Ulla Siltaloppi for their always kind help with practical matter.

Warm thanks to all my friends from Tampere for filling my time with wonderful parties and unforgivable moments. Special thanks go to Ada, Alex, Anca, Betty, Irina, Mircea, Radu for the special moments we spent together.

I wish to express my deep gratitude and warm thanks to my family and to my wife's family for unconditional support, understanding and love within the years. Special thanks go to my parents, Cecilia and Octavian, and my grandmother, Irina, for always being near me, despite the physical distance that has separated us most of the time during these years. Special thanks go to my wife Simona, for always painting the life in light and vivid colors, in those moments when everything seems in vain and everything fades out.

# Main abbreviations

ADSL	Asymmetric Digital Subscriber Line
API	Application Programming Interface
BIFS	BIInary Format for Scenes
CD	Compact Disk
DAI	DMIF Application Interface
DCT	Discrete Cosine Transform
DiffServ	Differentiated Services
DMIF	Delivery Multimedia Integration Framework
DRM	Digital Rights Management
DSM-CC	Digital Storage Media - Command and Control
DVB	Digital Video Broadcast
DVB-C	Digital Video Broadcast – Cable
DVB-S	Digital Video Broadcast – Satellite
DVB-T	Digital Video Broadcast – Terrestrial
DVD	Digital Versatile Disk
DVR	Digital Video Recorder
EDF	Earliest Deadline First
ES	Elementary Stream
FEC	Forward Error Correction
FS	Filesystem
HTTP	Hypertext Transfer Protocol
IDCT	Inverse Discrete Cosine Transform

IETF	Internet Engineering Task Force
IntServ	Integrated Services
IOD	Initial Object Descriptor
iTV	interactive TV
IP	Internet Protocol
JMF	Java Media Framework
LAN	Local Area Network
LNB	Low Noise Block
MPEG	Moving Pictures Experts Group
nVoD	near Video on Demand
OD	Object Descriptor
PCI	Personal Computer Interface
PCR	Program Clock Reference
PID	Packet Identifier
QoS	Quality of Service
RAID	Redundant Array of Inexpensive Disks
RGB	Red Green Blue
rpm	rotations per minute
RSVP	Reservation Protocol
RTCP	Real Time Control Protocol
RTD	Round-Trip Delay
RTP	Real Time Protocol
RTSP	Real Time Streaming Protocol
SAP	Session Announcement Protocol
SDP	Session Description Protocol
SIP	Session Initiation Protocol
SL	Sync Layer

SNMP	Simple Network Management Protocol
SSL	Secure Socket Layer
STB	Set-Top Box
TCP	Transport Control Protocol
TRIF	TRIVial File format
TS	Transport Stream (MPEG-2 Transport Stream)
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
URL	Uniform Resources Locator
VCD	Video CD
VDSL	Very high speed Digital Subscriber Line
VoD	Video on Demand
VoIP	Voice over IP
WLAN	Wireless LAN
WM	Microsoft Windows Media
xDSL	one of the Digital Subscriber Lines standards (ADSL, VDSL, etc.)
XV	XVideo

# List of Figures

1.1	The components of the considered VoD system and the chapters describing these components. . . . .	21
1.2	The use of the RTSP, RTP, SDP library (the communication infrastructure) by other system components. The organization of the thesis and articles sustaining the described work. . . . .	22
2.1	Interfacing the RTSP, RTP and SDP library. . . . .	27
2.2	The architecture of the integrated RTSP, RTP and SDP library. . . . .	31
2.3	Functions handling RTSP methods. The interface with the application of the RTSP, RTP and SDP library. . . . .	35
3.1	DMIF architecture and concept [30, 43]. . . . .	42
3.2	FlexMux and TransMux Channels in DMIF. . . . .	46
3.3	Mapping of ServiceAttach. . . . .	54
3.4	Mapping of ChannelAdd. . . . .	56
3.5	Mapping of UserCommand. . . . .	58
3.6	Mapping of ChannelDelete. . . . .	60
3.7	Mapping of ServiceDetach. . . . .	61
3.8	Previous version of ServiceAttach mapping to RTSP. . . . .	70
3.9	MPEG-4 System based on a server with RTSP-based DMIF instance. . . . .	72
3.10	MPEG-4 System based on a server with RTSP-only signalling. . . . .	73
4.1	Simplified player architecture. . . . .	81
4.2	Detailed player architecture. . . . .	82
4.3	Player states and transitions between states. . . . .	83
4.4	Media playback using software decoding plugins. . . . .	86
4.5	Media playback using hardware aided decoding plugin. . . . .	88
5.1	Stripping a file across a RAID0. . . . .	94
5.2	Results of continuous read. . . . .	100
5.3	The influence of the cache system on the Disk Pump performance (target: 400 Mbits/s, 10 seeks/second, 128 KB block size) . . . . .	101
5.4	The influence of the cache system on the Disk Pump performance (target: 400 Mbits/s, 10 seeks/second, 1024 KB block size) . . . . .	101
5.5	Total throughput of the Disk Pump algorithm - block size dependency. . . . .	102
5.6	The architecture of the VoD server. . . . .	103

6.1	Components of the multimedia system. . . . .	108
6.2	Components of the Management Platform. . . . .	112
6.3	Screenshot of IP TV Server configuration: Assigning satellites to DVB-S cards. . . . .	115
6.4	Screenshot of IP TV Server configuration: Assigning Channels to DVB-S Cards. . . . .	115
6.5	Screenshot of the User Gateway: Available Movies. . . . .	116

# Chapter 1

## Introduction

Video on Demand (VoD) is one of the main applications of the broadband multimedia. As concept, VoD is very similar to the movie rental paradigm: customers rent the content (movie) and watch it when they like. In a VoD system, the customers select and rent the movie from their home, with the TV and a remote control. The movie starts immediately after the selection.

A VoD system typically consists of several VoD servers, communication links and many VoD clients. A VoD server is able to receive requests from clients and after authentication and perhaps billing, it streams the selected content to the clients. A VoD client is typically a Set-Top Box (STB) connected to broadband network (e.g. to the Internet, by using one of the Digital Subscriber Line (xDSL) technologies). The STB is also connected to TV and speakers, these ensuring its audio-video output. One of the most important components in a VoD system is the middleware which ensures the management of the system and facilitates the movie selection by the customers.

### 1.1 Problems in current VoD systems

Several years have passed since Video on Demand (VoD) over IP protocol started being a research topic. Many times during these years it has been envisioned that VoD is only few years away and it would become as common as VHS or DVD players. A mature VoD service should replace VHS and DVD rentals completely, by offering more convenience, larger movie selection and better ways to choose the desired movies. Even after so many years of VoD research, VHS and DVD rental shops are much more common than VoD services. The reason behind this are some technical and economical problems related to VoD. The current technical problems of VoD make very difficult to find a valid business model for offering VoD services to mass markets.

The general scope of this thesis is to make some steps towards bringing high-quality, VoD multimedia content to people's TVs by solving some of the remaining technical problems of VoD systems. Some of the remaining challenges can be identified as being related with:

- **Insufficient interoperability between different components of the VoD**

**system [141].** A mature VoD system should contain many servers and many clients. It is very likely that such a mature VoD system will not be built at once, but it will be gradually scaled from a small system to a large system. Such a VoD system is then likely to be heterogeneous, containing several types of servers and several types of clients. It is generally expected that the newer components (servers or clients) are based on more advanced technology; however, they should be still compatible with the old components. If these compatibility issues are solved, the old components can also be replaced gradually with newer components. Ideally, the system should be designed in such a way that all clients can communicate with all servers.

- **High price and complexity of the VoD system [124, 22, 90, 69, 70, 95].** As demonstrated by several trials in the past, the use of a VoD service is very sensitive to price. A price for a VoD movie that is significantly higher than the price of renting the same movie from a DVD rental shop would prevent the VoD service to be widely adopted. The components of a VoD system (servers and clients) traditionally have a complex internal architecture and thus, a high price tag associated with them.
- **Lack of players for multimedia content on certain platforms [124].** A mature VoD system would allow its customers to choose between several types of clients (e.g. Set-Top Boxes), each of these clients having different capabilities. All these clients will have to be able to play the VoD content (their primary purpose), but some may allow, for example, playing DVDs, games, e-mail, web browsing. It is also very important that all the VoD clients remain cost-effective. One possibility to create such cost effective clients is to create VoD-capable applications on platforms (e.g. Linux) already allowing all the other applications.
- **Insufficient VoD System management [55].** Maintaining a VoD system is very important in terms of price (to have a low total cost of ownership), adding new content, allowing and integrating other services into the system and overall system availability. Such a management system should be able to configure, control and monitor in a cost effective way a heterogeneous group of servers or other components.
- **Lack of proper customer interface to the VoD system.** It is very important that the VoD service has an easy-to-use, comprehensible interface with the customers, otherwise not many people are going to use the service. This interface has to allow a very simple selection of the movie to be played, and advanced selection methods (e.g. based on feedback from users) should be built around it.
- **Possibility of unauthorized use of the VoD content [74, 136].** Content owners fear that unprotected VoD content may be easily recorded by some users on devices outside the VoD system. Once the content is recorded, it can be used in several unauthorized ways (e.g sharing it over the Internet). Content owners

may lose revenue because of potential for paying customers getting the content from the Internet and not from authorized distributors (e.g. the VoD service). The majority of the content owners would not license content to VoD services unless a solution is found that would prevent the above scenario.

- **Absence of Quality of Service (QoS) for the transport of multimedia content over the network [17, 16, 7].** The vast majority of the Internet services available today do not guarantee any delivery parameters for the data traffic. It is possible and likely that some packets are going to be lost on their way from the server to the clients. The loss of these packets may degrade significantly the perceived quality of the streamed movie.

This thesis gives solutions for some of the above mentioned problems by proposing new architectures at several layers of the VoD system. The thesis describes also efficient implementations of the proposed architectures. The main purpose of the presented implementation work is the validation of the presented architectures.

## 1.2 Problems outside the scope of this thesis

Two from the problems of VoD systems identified above are left outside the scope of this thesis: preventing unauthorized access to content and improving the QoS.

Good technical solutions to prevent unauthorized use of content do exist. Today, they are mainly used for TV content over DVB (Satellite, Terrestrial, Cable). These solutions consist of encrypting the TV content when sent over the DVB medium. The content is decrypted at the playback end by a special decrypting module, if it has the permission to do so. This solution can be extended to TV content delivery over other mediums (e.g. IP) and also to VoD. Another technical solution for content protection in VoD systems is DRM. This solution also encrypts the content, but also specifies in details what the user can do and can not do with the content.

Even if technical solutions for content protection do exist, there is still fear among content providers that some users may bypass content protection schemes and release the unprotected content to the world. Bypassing content protection schemes by unauthorized users (known also as piracy) has always existed, but the damage these users could do was limited. These users (pirates) were limited to watch the content themselves or to record part of it on tapes and share it with few friends. Today, the existence of peer to peer networks (p2p) makes possible that one user can share unprotected content with the world, thus increasing the damage (loss of revenue) for content owners. In this respect, we can state that the content protection mechanisms are as good as ever, it is just the sharing possibilities of the content that have skyrocketed in the recent years. There is one exception to this, namely the DVD content protection, which was broken in the year 1999. The existence of software tools makes that nowadays, one can gain unauthorized access to a DVD content (DVD rip) with only a common computer and few pieces of software.

The content owners fear that an encryption scheme that can not be broken today, can be broken in the near future. Breaking an encryption scheme may not be an

entirely technical thing, but it can be the result of several factors: the breaking of the DVD content protection was much eased by the fact that one licensed decryption key (from Xing) was stored unencrypted on DVDs. The damage that can be done to content owners revenues by breaking the encryption schemes is augmented by the vast possibilities of sharing the pirated content.

Even if there would be an encryption scheme that is mathematically proved unbreakable, it would be still prone to human handling errors, thus breakable. This is the main reason the author of the thesis considers that today, preventing unauthorized access is a technically solved, but economically and business unsolved problem. Because the author considers the technical problem solved, preventing unauthorized access to VoD content is not included among the research topics covered by this thesis.

The author also believes that the absence of a proved solution for the prevention of unauthorized use is not the main factor that prevents VoD commercial deployments today. Guarding VoD content against unauthorized use is technically similar with guarding IP TV content. However, there are much more IP TV services than VoD services for consumers in Europe. These IP TV services sometimes have low (or inexistent) protection against recording them in clear, at least for some channels. This proves that, at least in certain cases, content protection is not the main concern in the design of a digital media streaming system.

The problem of Quality of Service in the Internet has been researched for many years and technical solutions have been found at several protocol layers. However, in the real Internet, for the vast majority of Internet users there is no QoS service. The reason for this is economical: providing QoS would be closely related with billing. This would require trade agreements between all Internet Service Providers and some kind of billing and revenue splitting methods between service providers involved in one transaction. One such transaction can be the streaming of a movie, that involves 20 service providers. Currently it is very difficult to define fair revenue splitting between so many service providers.

Several important error correction mechanisms are available for a VoD system, but they have to be implemented at the application layer. They include Forward Error Correction (FEC), packet retransmission, transmission of several layers of quality, server encoding adaptation based on feedback from client (e.g. using RTCP). While application layer error correction may improve significantly the user perceived quality of the received stream, it can not guarantee any performance level.

This thesis concentrates on basic architecture aspects and does not deal with QoS or error correction, because of two main reasons:

1. The VoD service is supposed to be offered from only one local network service provider. Because of this we can assume that the network link between the client and the server is able to accommodate the traffic, or, existing QoS solutions (Differentiated Services - DiffServ or Integrated Services - IntServ) can be employed.
2. The implementation of a QoS solution between several Internet Service Providers is technically solved, but economically unsolved.

If further development added error correction to any of the architecture presented in this thesis, FEC or packet retransmission based on RTCP feedback would be suitable to be added to the joint RTSP, RTP and SDP library described in Chapter 2.

### 1.3 Solutions proposed in this thesis

There are several types of solutions for each of the problem listed in the section 1.1.

The problem of interoperability between different components of a VoD system comes from the vertical approach used in the past in building VoD systems. In this approach a vendor provides the whole VoD system and ensures that the components are interoperable. When a component from a different vendor is brought into the system, it is likely that it will not interoperate with the rest of the system.

Transport and control of VoD multimedia content have been standardized at the application level by two big standardization groups: Moving Pictures Experts Group (MPEG) and Internet Engineering Task Force (IETF). The MPEG group standardized the transport and control of MPEG-2 content into a standard called Digital Storage Media - Command and Control (DSM-CC) and it also standardized the transport and control of the MPEG-4 content into a standard called Digital Multimedia Integrated Framework (DMIF). The IETF group standardized the transport and control of VoD content by creating three protocols: Real Time Streaming Protocol (RTSP), Real Time Protocol (RTP) and Session Description Protocol (SDP). In general, VoD system components using MPEG protocols are not compatible with system components using IETF protocols.

In Chapter 3, this thesis proposes a solution to overcome this problem by mapping the DMIF primitives to the IETF protocols. In practice, this mapping allows a VoD system component using DMIF to interoperate with another system component that uses IETF protocols. For this, the mapping is implemented inside a DMIF plugin. The mapping, the architecture of the DMIF plugin, and the details about its implementation and performance are presented in Chapter 3.

The high price and complexity found in the components of VoD systems results from the need to combine several hardware and/or software sub-components together into a single VoD system component, in an integrated and reliable manner. Having many sub-components requires complex communication and data exchange methods between these sub-components. These communication and data exchange methods make the design of VoD system components a difficult and complex task.

A possible solution to this problem is to group sub-components into bigger units and split the general communication into two layers: the communications between sub-components inside a unit and the communication between units.

This thesis proposes such a solution to overcome the complexity of the networked communications between clients and servers inside a VoD system. In particular, the thesis proposes the architecture of a joint RTSP, RTP and SDP library that handles internally the communication between these three protocols and presents the application with a single interface for communication with other VoD system components (clients or servers). The architecture of this joint RTSP, RTP and SDP

library and details about its implementation, usage and performance are presented in Chapter 2.

The high price and complexity found in the components of VoD systems may also be triggered by the performance requirements. In order to ensure certain performance parameters, very powerful technologies might be needed, which can be highly expensive (in terms of price and complexity). One such case is the storage subsystem in a VoD server.

Multimedia data retrieval from hard-disks has posed great challenges to VoD server designers, because of its relative low speed compared with the speed of other server components. This low speed inquires additional penalties when changing the read position from a hard-disk due to the time required to perform the seek operation. Traditionally, the storage subsystem of a VoD server consists of expensive hard-disks that have very good (best) individual performance, thus influencing the overall performance of the storage subsystem.

An alternative to this approach is to create a performance model that, given the parameters of the storage subsystem and the resources in the system, approximates what can be expected from the storage subsystem in terms of number of simultaneous streams. Such a model would allow a VoD server designer to actually compute the trade-off between the cost of the components used in the storage subsystem and the overall performance of the server.

In Chapter 5, this thesis proposes such a model to compute the expected performance level of a storage subsystem. The validity of the performance levels predicted by the model are verified experimentally. We also show that very good performance levels can be obtained not only by using expensive, top-of-the range components, but also by using commodity components.

The lack of multimedia players on certain platforms can be solved by designing a player for the platform or by porting a player from other platforms. Designing a future-proof multimedia player to be used in a VoD system is not a trivial task and requires careful analysis and design.

In Chapter 4, this thesis proposes the design of a VoD multimedia player for the Linux platform. The validity of the design is verified in practice by implementing the player.

The need of good system management in a VoD system comes from the need of cost-effectively maintaining such a system. Post-deployment operations include content management (uploading to servers, deleting from servers), servers configuration, servers monitoring and servers control. The management system should ensure the reliability of the overall VoD system and allow upgrading different individual components (servers, clients).

In Chapter 6, this thesis proposes a solution to the system management problem that consists of a web-based management platform capable to configure, monitor and control several types of servers (including VoD servers). The management framework is designed based on an extensive set of requirements. The architecture of this management framework, and the details about its implementation are presented in Chapter 6.

The need of a good customer interface in a deployed VoD system is crucial, because

the usability of this user interface may retain customers or discourage them from using the system.

In Chapter 6, this thesis proposes the architecture of an interface that acts as a media portal for customers. This interface is designed and implemented together and as a part of the management framework presented in Chapter 6.

## 1.4 Publications on which the thesis is based

This work is partly based on the following publications (listed according to their reference numbers from the Bibliography):

- [12] A. Basso, S. Varakliotis, R. Castagno, and F. Lohan. Transport of MPEG-4 over IP/RTP. *European Transactions on Telecommunications*, 12(3), May–June 2001
- [20] R. Castagno, S. Kiranyaz, F. Lohan, and I. Defée. An Architecture Based on IETF Protocols for the Transport of MPEG-4 Content over the Internet. In *Proc. of International Conference on Multimedia and Expo (ICME)*, pages 1322–1325, New York, USA, August 2000.
- [76] F. Lohan and I. Defée. Modularity in open media terminal system architecture. In *Proc. of International Conference on Multimedia and Expo (ICME)*, pages 708–711, Tokyo, Japan, August 2001.
- [77] F. Lohan and I. Defée. Open media terminal system. In *Proc. of International Conference on Media Futures*, pages 153–156, Florence, Italy, May 2001.
- [78] F. Lohan and I. Defée. Integrated web-based management system for a heterogeneous multimedia system. *Recent Advances in Communications and Computer Science*, WSEAS Press, pages 392–397, 2003.
- [80] F. Lohan, I. Defée, R. Castagno, and S. Kiranyaz. Content Delivery and Management in Networked MPEG-4 System. In *Proc. of European Signal processing Conference (EUSIPCO)*, volume 4, pages 2313–2316, Tampere, Finland, September 2000.
- [81] F. Lohan, I. Defée, and H. Hakulinen. Design and Implementation of an Open Broadband Multimedia System. In *Proc. of Packet Video Workshop (PV)*, pages 246–255, Kyongju, Korea, April–May 2001.
- [82] F. Lohan, I. Defée, and H. Hakulinen. Networked Multimedia System Based on Open Architecture. In *Proc. of International Conference on Consumer Electronics (ICCE)*, pages 344–345, Los Angeles, USA, June 2001.
- [83] F. Lohan, I. Defée, and M. Vlad. The Architecture of an Integrated RTSP, RTP and SDP Library. In *Proc. of International Conference on Telecommunications (ICT)*, pages 338–342, Bucharest, Romania, June 2001.

- [85] F. Lohan, I. Defée, M. Vlad, A. Pop, and P. Sastry. Integrated System for Multimedia Delivery Over Broadband IP Networks. *IEEE Transactions on Consumer Electronics*, 48(3):564-574, August 2002
- [86] F. Lohan, P. Sastry, and I. Defée. Broadband Network Set-Top Box System. In *Proc. of Protocols for Multimedia Systems (PROMS)*, pages 257-263, Krakow, Poland, October 2000
- [87] F. Lohan, M. Vlad, and I. Defée. Analysis and optimization of disk retrieval techniques in a PC-based VoD server. In *Proc. of the 7th WSEAS International Multi-conference on Circuits, Systems, Communications and Computers*, Corfu Island, Greece, July 2003.

These publications are based on the research work performed within several industrial projects in which the author was a member of the team working for the Tampere University of Technology. None of the above publications has been used as a part of any other dissertation.

## 1.5 Outline of the thesis and main results

This thesis is based on research performed in several projects in EU, National Technology Agency of Finland, and industry. The structure of this thesis was chosen with the intention to provide a comprehensive description of architectural-based solutions to some of the problems in VoD systems. Each chapter describes issues related with the architecture and implementation of VoD components. The order of the chapters is chronological; each chapter is based on work and experience described in previous chapters. The VoD system considered in this thesis has four basic components: Servers, Clients, Communication Infrastructure and a Management Platform. Fig. 1.1 shows the relation between these components and enumerates the chapters in which different components are described. Fig. 1.2 gives another view about the organization of this thesis and also presents the publications each chapter is based on.

Chapter 2 describes the architecture of a joint communication library that uses RTSP, RTP and SDP protocols. This library ensures the remote retrieval services for all servers and clients described in this thesis, as seen in Fig. 1.2. The work presented in this chapter is based on [83].

The joint RTSP, RTP and SDP library creates a package, presenting the applications using it with a single, consistent interface. This package hides the internal communication between the RTSP, SDP and RTP protocols from the rest of the application. This integrated package is a good starting point for completely hiding the delivery layer from the application. The second main benefit of the library is the fact that it gives the application the possibility to extend the RTSP and SDP protocols, as allowed by the standards, without modifying the library itself. Extending RTSP and SDP is done by overloading two pure virtual classes. The benefit of this approach is the preservation of the architecture of the library in the extension process and the preservation of the quality of the implementation of the library.

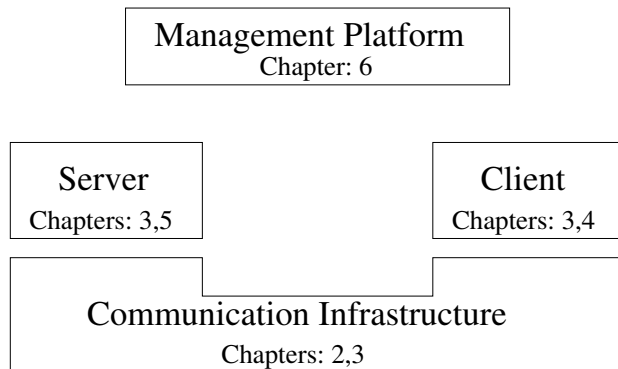


Figure 1.1: The components of the considered VoD system and the chapters describing these components.

Chapter 3 describes the architecture of a system for remote retrieval of MPEG-4 content based on IETF protocols. The work presented in this chapter is based on [20, 80, 12].

The scientific contribution the approach is the mapping between DMIF and the IETF protocols. As a consequence of this mapping we achieve interoperability between VoD components based on IETF standards and VoD components using DMIF. This interoperability allows the leverage of some existing knowledge and experience from existing systems in the new, emerging, MPEG-4 based systems.

Chapter 4 describes the architecture of a media player for high-quality VoD content. This player was designed and implemented for both STB and PC environment based on the Linux platform. The work presented in this chapter is based on [86, 81, 82, 77, 76, 85].

This player was one of the first Linux players having a modular architecture and being capable of playing MPEG-2 Transport Stream (TS) from a remote server.

Chapter 5 describes the architecture of a VoD server initially created to test the media player described in Chapter 4. In time, this server evolved as a high-capacity VoD server due to an optimized disk retrieval algorithm. The main part of Chapter 5 contains details about this algorithm and experimental data validating it. The work presented in this chapter is based on [85, 87].

The scientific contribution of this work consists in demonstrating that it is possible to achieve the high performance required by a VoD server when reading multimedia streams with a PC running Linux and having a storage subsystem based on Redundant Array of Inexpensive Disks (RAID). Our maximum achieved performance was 500 Mbits/s. This speed is enough to serve more than 100 clients requesting MPEG-2 streams (at 4 Mbits/s). The tests show that global system performance is highly correlated with the hardware and software configuration, and a correct tuning process is required in order to achieve the optimum performance.

Chapter 6 describes the architecture of a management platform for configuring, monitoring and controlling several types of servers. This platform imposes certain

# VoD System

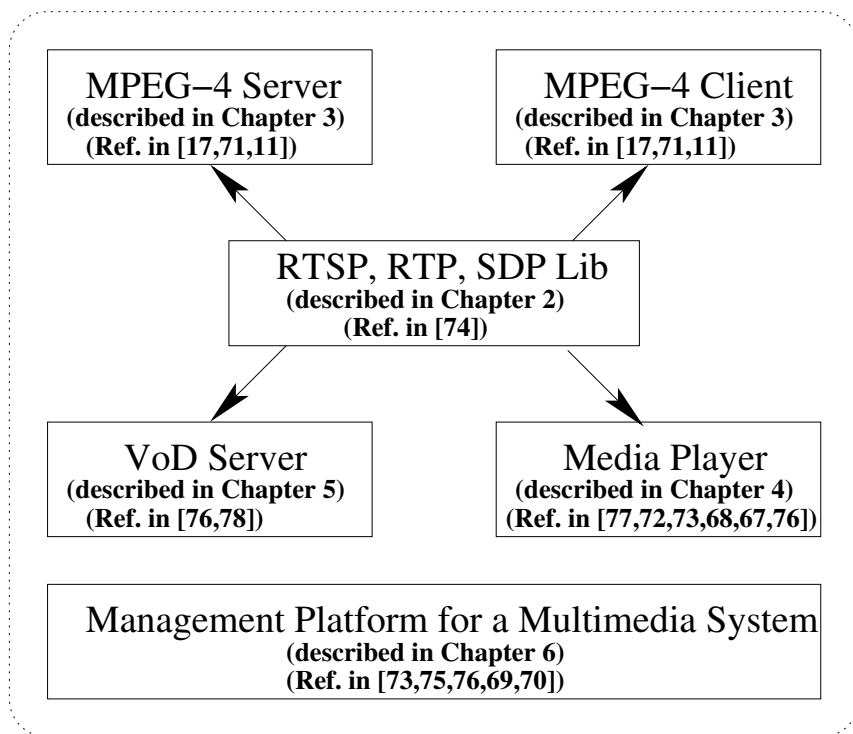


Figure 1.2: The use of the RTSP, RTP, SDP library (the communication infrastructure) by other system components. The organization of the thesis and articles sustaining the described work. .

guidelines for the other components of the multimedia system. It thus creates the framework of a VoD system that has the capability to accept many heterogeneous components. The work presented in this chapter is based on [82, 84, 85, 79, 78].

The scientific contribution brought by this management platform is its design, capable of managing several different servers, capable of ensuring their reliability and capable of presenting the end-users with an attractive user interface to the content available from the system.

## 1.6 Overview of existing solutions to the problems discussed in this thesis

This section gives some details about VoD solutions available from important players in the VoD industry. From the point of view of this thesis, we are interested in solutions adopted on the server and client sides of a VoD system. There are several companies providing VoD servers, including Kasenna, nCube, SeaChange, Concurrent, Real, Apple, Envivio, iVast. On the client side, solutions are provided by

Microsoft, Real, Apple, iVast, Amino, Kreatel, Pace.

The problem of compatibility between VoD servers and clients arises from the fact that different server brands use the standard control and transport protocols in non-standard ways. Server companies have added their own fields, commands or command order to the RTSP protocol, and they expect the client to use these modifications. Such an example consists of replacing the recommended command sequence at initialization in RTSP (DESCRIBE followed by SETUP and PLAY) with a single command (SETUP). Some servers do not support the standard (generic) sequence order of commands.

STB manufacturers are dealing with this situation by supporting several RTSP “profiles”. Each of these RTSP profiles works with a brand of servers, or with other brands that clone the RTSP behavior of that brand. Changing the RTSP profile on a STB is usually difficult and requires accessing the set-up menu, or even loading some code modules on the STB. Such operations can only be done by the operator. This makes that one STB can support only one type of VoD server. The disadvantage of this solution, for the service provider, is that, once they start with a VoD server vendor, they are locked to that vendor, since the deployed STB are not able to handle the second type of VoD server simultaneously. Commercially, this solution locks the customers (operators and service providers) to a single VoD server brand, or to clones of this brand.

Our solution proposed in Chapter 2 allows a STB to support several RTSP profiles without the need to change any settings. This is done by detecting the type/vendor of VoD server and using the specific set of RTSP commands, fields and command order that the server expects. We actually provide a single point of entry to several RTSP “dialects”. Our solution does not consist in detecting the VoD server type and then loading the right RTSP code, but rather integrating all the known RTSP profiles in a single package. At runtime, after detecting the server, the client selects the profile-specific features and rules and uses them in further communication with the server.

In Chapter 3 the interoperability problem and its solution are extended to MPEG-4. Currently, commercial solutions that provide MPEG-4 based VoD only use the audio and video compression standards from MPEG-4. The system part (BINARY Format for Scenes - BIFS, Object Descriptor - OD) and DMIF (Delivery Multimedia Integration Framework) are not used. The Internet Streaming Media Alliance (ISMA [59]) provides standards for carrying MPEG-4 content over RTP and RTSP. This seems to have solved any potential interoperability problem between MPEG-4 clients and MPEG-4 servers. There is also an open-source, ISMA-compatible server that allows VoD client manufacturers to test the compatibility of their solutions. This server is called Darwin Streaming server and it is available from Apple, one of the founders of ISMA.

However, even if it seems to solve the basic interoperability problem in MPEG-4 VoD systems, ISMA does not specify any use of DMIF. The solution proposed in Chapter 3 would allow a DMIF-based client to use a non-DMIF-based server (e.g. an ISMA-compliant server).

The problem of server incompatibility becomes even bigger in a heterogeneous

system that has both MPEG-2 and MPEG-4 servers, where a client (supporting both encodings) would need to interoperate with all servers in the system. The solution proposed in Chapter 3 is able to solve this problem by considering the RTSP command sequence for requesting an MPEG-4 file as being just another RTSP profile. In this way the client can request a stream (URL) without even knowing what type of encoding (MPEG-2 or MPEG-4) is used, or even what type of server hosts the stream. The solution proposed in Chapter 3 arguably requires minimal modifications of the transport and control part in a client (that uses DMIF or not), in order to make it compatible with several MPEG-2 and MPEG-4 servers.

The problem of high complexity and price of VoD servers comes from the fact that some of the VoD vendors use less-common or sometimes proprietary hardware and software. This makes that the customers of these VoD servers have to rely on the VoD server provider for any non-trivial maintenance of the server. Any spare part may need to be bought from the server vendor and any upgrade to the server software (operating system, VoD server or other necessary applications) may require intervention from the server vendor. This makes the maintenance costs of these VoD servers very high.

The solution proposed in Chapter 5 is platform and operating system independent. This solution allows its implementation on any operating system. Chapter 5 describes certain optimizations that are specific to Linux, but it is possible that this type of optimization to be available to other operating systems as well. Having such a platform-independent solution reduces considerably the maintenance cost of the server. If implemented on PC and Linux, the customers would be able to maintain (at least partially) the hardware and the operating system by themselves, because there is enough knowledge about PC and Linux in many companies. In this case, the VoD server vendor would only be required to maintain its own software product.

The lack of multimedia players on certain platforms was observed few years ago, for the Linux operation system: there was no player capable to play MPEG-2 streams from the network. At that time, Linux became a very attractive platform for the STB manufacturers, because it provides a good infrastructure and several useful applications at no licensing costs. Linux became attractive because of the high price-sensitivity of the STB market and because it become possible to build a cost-effective STB capable of running Linux. In current STB, the video (MPEG-2) decoding and playback is done in hardware. The MPEG-2 stream is acquired through an RTSP library and fed to a demultiplexer (possibly software-based). The demultiplexed video elementary stream is then fed to the hardware decoder, who also takes care of the video display on screen, by using alpha blending.

The player proposed in Chapter 4 was one of the first Linux players to support STB and hardware-aided decoding. It also brings hardware-aided playback and software-only playback under a single architecture, which is something unique, probably even today.

The problem of VoD servers management is currently solved by the VoD vendors by providing web-based and/or Java based management solutions. These solutions are usually capable of managing one or many VoD servers, but of the same type.

The management platform proposed in Chapter 6 allows the web-based manage-

ment of several types of servers, not only VoD. This way it is possible to manage a complete, heterogeneous system for multimedia content delivery to clients using different servers and methods (VoD, TV over IP).

The problem of customer interface to the VoD systems is currently solved by different companies than those supplying the servers and clients in a VoD system. These companies usually provide a web-based interface that can be accessed using a remote control. None of the customer interface solutions seem to be integrated with the management system of the VoD servers.

The solution proposed in Chapter 6 is web-based, tightly integrated with the server's management system. In this way, the availability of new movies and channels is automatically handled by the customer interface, by accessing the same database as the management system.

## Chapter 2

# Integration of Communication Protocols

One of the main applications of networked multimedia is remote streaming content retrieval. It is based on the client-server model, where the client is playing content sent over the network from the server. There are three main aspects of remote content retrieval:

- the control of the content, that transmits the commands issued by the client (e.g. Start, Play, Pause, Stop, Fast Forward) to the server.
- the transport of the streaming content from the server to the client. Typical network bandwidth involved ranges from hundreds of Kbits/s to several Mbits/s, for a single stream.
- retrieval by the client of information associated with the requested content (e.g. MPEG-2 movie): its length, audio and video encoding, author and related metadata, transport parameters.

These three aspects of remote content retrieval are independent of each other, thus it makes sense to have a separate protocol for each of them, but at the same time they operate together, so a complete solution is needed to solve all the three aspects of remote control retrieval. It is important to have a solution based on standards, because this is the easiest way to ensure that clients and servers made by different parties will interoperate together.

IETF has created standard protocols for each of the three aspects of remote content retrieval, for the case when the underlying network is based on the IP protocol. Two standardization groups were created, the AVT (Audio Video Transport) group [37], which deals with content transport protocols, and the MMUSIC (Multiparty Multimedia Session Control) group [38], which deals with protocols related to media control. The result of the AVT group standardization effort includes the Real Time Protocol (RTP) [119]. The RTP standard provides specifications for media transportation from the server to client using data packets. For data transport RTP uses standard UDP or, sometimes, TCP Internet protocols. RTP may be accompanied by

the Real Time Control Protocol (RTCP) that sends feedback information from the client back to server [119].

The MMUSIC group’s standardization effort includes the Real Time Streaming Protocol (RTSP) [118], used for content control, and Session Description Protocol (SDP) [49], used for retrieval of information about the content. RTSP is used by the client to send commands to the server, and also sends back the server’s response. The SDP protocol is used inside RTSP, as data attachment to a command (in the **DESCRIBE** command) in order to send to the client initialization information related to the desired multimedia stream. This information should contain the number of individual media streams making the multimedia content, the transport method for each individual media stream and the identification of the compression standard. The RTSP was designed to rely on TCP as IP transport protocol, but it also can work over UDP. For multimedia data transport, RTSP was designed to rely on RTP, but it can accommodate other transport protocols, too.

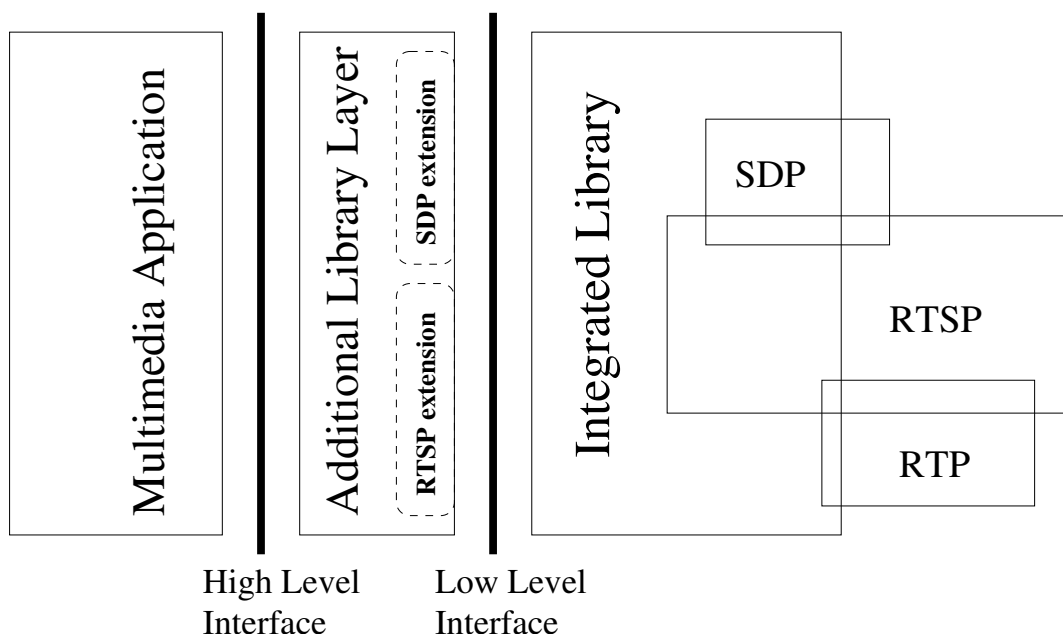


Figure 2.1: Interfacing the RTSP, RTP and SDP library.

Each of these three protocols can be implemented and used completely independently from each other. However, a complete IETF-based remote content retrieval solution should at least implement all three protocols. There are several implementations of RTSP, SDP [117] and RTP [116] protocols available, but they are implemented more or less independently of one another. This makes the handling of remote content retrieval difficult for the application, because it has to manage the three protocols separately. The application has to know and to take into consideration protocol details, such as the RTP profile or SDP information about URI (Uniform Resource Identifier) addresses controlling individual streams in a session. Ideally, the media delivery should be totally transparent for the multimedia application, but the ne-

cessity to manage three independent protocols makes such separation between the application itself and the delivery mechanism very difficult.

This problem is addressed in this chapter by describing an integrated implementation of all these three protocols into a single library (package). This library provides the application with a single, complex but yet comprehensive interface that hides unnecessary protocol details from the application. This interface is powerful enough to allow still for a good control over the protocols involved.

Many remote retrieval applications also implement extensions to standard protocols [20, 86, 12, 85]. In order to achieve interoperability with these applications, extensions to RTSP and SDP are required. Our library solves this problem by embedding extension capabilities into the library. These capabilities are accessed through the library Application Programming Interface (API), so that protocol extensions are handled outside the library. In this way, the library itself is not modified when a new extension is added.

Because of its complex API, creating a second layer, providing a much higher API to an application, is the preferred way of using the RTSP, RTP and SDP library. This second layer can also handle possible RTSP or SDP extensions (Fig. 2.1).

## 2.1 Short presentation of RTSP, SDP and RTP protocols

The RTSP protocol [118] is an application-level protocol for controlling the delivery of data with real-time properties. It provides an extensible framework to enable controlled, on-demand delivery of real-time data, such as audio and video. Sources of data can include both live data feeds and stored clips. This protocol is intended to control multiple data delivery sessions and provide a means for choosing the protocol stack for the transport channels. This protocol stack usually specifies three protocols: the lower transport protocol (e.g. UDP or TCP), the high level transport protocol (e.g. RTP) and the profile for the high level protocol (e.g. RTP profile for MPEG-2 [54]).

RTSP consists of exchange of text messages between the client and the server. The client sends request messages, and the server answers to each request (similar to HTTP [29]).

The protocol defines the set of methods (commands) for the communication between the client and the server. Each method has several fields for communicating additional information. The most important methods are: **DESCRIBE**, **SETUP**, **PLAY**, **PAUSE** and **TEARDOWN**.

The **DESCRIBE** method is intended for retrieving initialization information from the server about the media session that is going to be played. This initialization information contains the number and type of individual media streams composing the media session and information about the available transport methods on the server side. In this way the client can decide what media streams it wants to play (e.g. if there are several audio language streams the client can choose only one of them). The client can also decide upon the preferred/supported protocol stack (e.g. media may be streamed using plain UDP or using RTP over UDP and the client can

choose either of these transport methods). The initialization information may also contain some meta-data about the author of the media, the duration of the media and information if individual streams can be controlled separately or not. All this initialization information can be attached to the **DESCRIBE** response in the SDP format [49]. The SDP protocol is also text based and the information is split in several sections: the global section that contains information about the session and the media sections, one for each media available in the session, containing information about that media.

After receiving the initialization information, the client can request the individual media streams available in the media session. A movie may consist of two streams, an audio stream and a video stream, so the client may need to request both media streams in order to play the movie.

The request for individual media streams is performed via the **SETUP** method. The most important information field is the Transport field that contains information about the data transport possibilities of the client. The answer from the server also contains a Transport field that actually chooses one of the transport methods proposed by the client.

Example:

```

Client->Server  SETUP      rtsp://video.ex.com/video RTSP/1.0
                  CSeq:      3
                  Transport: RTP/AVP/UDP;multicast;t1=127;mode="PLAY",
                              RTP/AVP/UDP;unicast;client_port=3456;mode="PLAY"
Server->Client  RTSP/1.0    200 OK
                  CSeq:      3
                  Session:    23456789
                  Transport:  RTP/AVP/UDP;unicast

```

In the above example, the client requests a media URI and advertises to the server two methods of data transport. The first advertised method in which the client is able to receive the stream is data multicast using the RTP protocol, Audio-Video Profile. The second method is data unicast, where the server sends RTP Audio-Video Profile packets over UDP, to client port number 3456. In the server's response we see from the Transport field that it chose the second method. The reason why the server chose the second method is outside the scope of RTSP and depends entirely on the server (e.g. the server is not multicast enabled).

In addition to the selection of the transport method, the server response to a **SETUP** method creates the Session Identifier (ID) for that RTSP session. This Session ID has to be present in all subsequent RTSP messages in that RTSP session. The Session ID is useful for RTSP sessions without permanent connection, as a method of identifying to what session subsequent RTSP messages belong to.

The CSeq field counts the Request-Response pairs (in our example it indicates that this is the third Request-Response pair in this RTSP session). The CSeq is useful for RTSP sessions without permanent connection, as a method of detecting lost requests or answers. The CSeq field is also useful to match responses to requests

for clients issuing several RTSP commands without waiting for answers.

After the media data stream is set up, the client has to send the **PLAY** command to the server in order for the server to start sending the data. Additional fields may specify the playing position and speed.

The **PAUSE** method interrupts the stream transmission.

When the client wants to end the media session, it sends the **TEARDOWN** command to the control URI of the session. **TEARDOWN** may also be applied to individual media streams.

The Real Time Protocol (RTP) provides end-to-end network transport functions suitable for applications transmitting real-time data, such as audio or video, over multicast or unicast network services. Data sent over RTP is fragmented and transmitted in packets, similar to the UDP case. RTP also adds its header to the packets. The RTP header contains several fields that specify some additional information. The most important fields for carrying multimedia streams, are:

1. The payload type - specifies what kind of data the RTP packet is carrying. For different data types, there are defined RTP profiles that specify additional information to be carried out in the RTP packet, as extension fields (header extension).
2. The sequence number - specifies the number of the RTP packet, so that the receiver can identify the order of the packets and also the packet loss, by counting the missing packets.
3. The timestamp - provides timing information that can be useful at the receiver for computing different network parameters (round trip delay, jitter), or for the decoding process, or for the displaying process.

There are also several IETF standards (RFCs) which define RTP profiles of how different types of media (e.g. MPEG-2 or MPEG-4) are encapsulated in RTP packets [54, 45, 46, 133, 66, 10, 130].

## 2.2 The architecture of the integrated RTSP, RTP and SDP library

Our goal was to develop a unified framework for integration of the RTSP, RTP and SDP into a single library package.

The library obeys the object-oriented paradigm. It is designed using objects as functional components and it was implemented in C++. The general architecture of the library resembles to a tree of objects (Fig. 2.2). The library is organized in two parts:

1. The messages handling. This part is responsible for generating and parsing the RTSP and SDP text messages. It defines two complex structures for keeping the information from an RTSP and respectively SDP messages. It also defines

functions for handling the RTSP and SDP messages, most important being the conversion between the text RTSP and SDP messages and their associated structure. The message handling part is composed from the *RTSPmsg*, *SDP*, *RTSPextension* and *SDPextension* objects (Fig. 2.2).

2. The RTSP session handling. This part is responsible for:

- (a) managing and keeping together resources associated with the RTSP sessions handled by the library. The *RTSPlib* object is responsible for this task (Fig. 2.2).
- (b) managing and keeping together resources associated with each RTSP session (e.g. network connection, command handling, stream creation). The *RTSPsession* and *RTSPinterface* objects are responsible for this task (Fig. 2.2).
- (c) managing and keeping together resources associated with each media stream (e.g. network resources, sending and receiving data, encapsulating and decapsulating data into and from RTP packets). The *RTSPstream* and *RTP* objects are responsible for this task (Fig. 2.2).

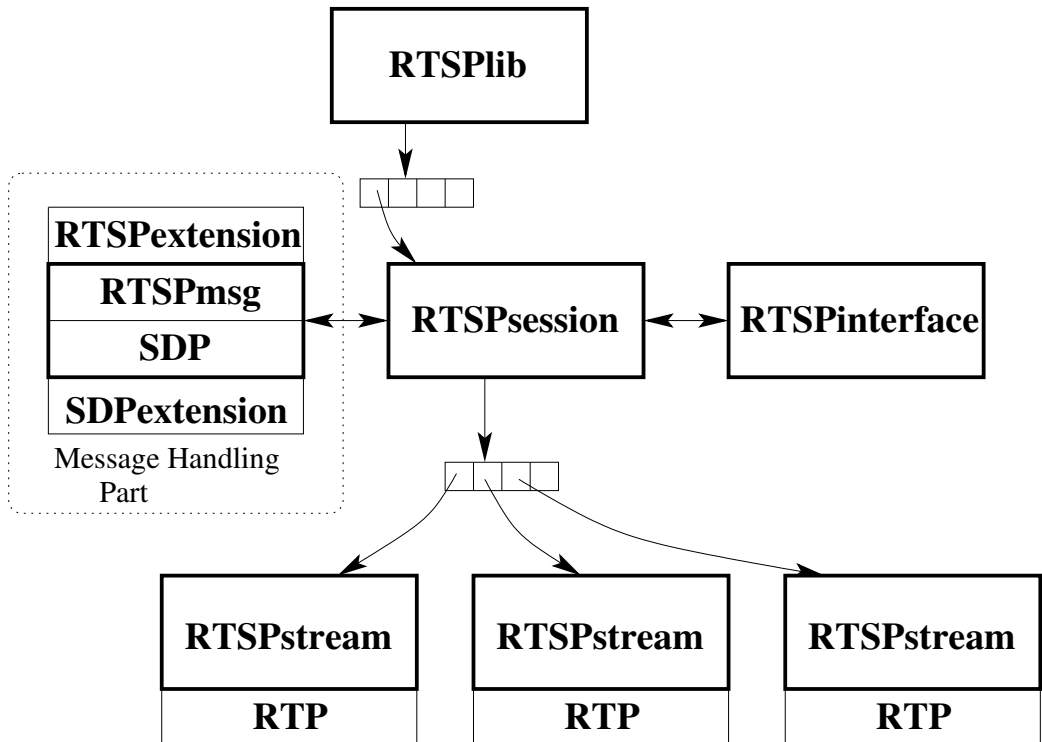


Figure 2.2: The architecture of the integrated RTSP, RTP and SDP library.

### 2.2.1 The messages handling part

Four objects handle the incoming and outgoing RTSP and SDP messages. The *RTSPmsg* object defines a complex structure able to keep all the information contained in an RTSP message in a more suitable fashion. When such a message is received by the library, the *RTSPmsg* object is parsing the message and fills in this structure. The library then deals only with the elements of the structure. When an RTSP message is to be sent, the library fills the elements of this structure and the *RTSPmsg* object creates the text message corresponding to the information in the structure.

The *RTSPextension* object handles the extension fields used to extend the RTSP protocol. The concept of this object is very similar to the *RTSPmsg* object: it has a structure able to keep the information contained in the extension fields. The *RTSPextension* object is part of the library's API. If the client application is not expecting to handle RTSP extension fields, the *RTSPextension* object is not needed. If the client application is expecting to handle RTSP extension fields, then the *RTSPextension* object should be implemented (the structure accommodating the information from the extra fields has to be defined and the parsing and message generation functions have to be implemented). When the library receives an RTSP message containing some fields that the *RTSPmsg* object is not able to handle, those fields will be forwarded for handling to the *RTSPextension* object, that will parse those fields and fill its associated structure. When sending RTSP messages containing extension fields, the client application has to fill the elements of the extension structure defined by the *RTSPextension* object. When the *RTSPmsg* creates the RTSP text message to be sent, it also calls the corresponding function from the *RTSPextension* object to generate the extension fields.

A similar approach is used for SDP messages. The *SDP* object defines a complex structure used to keep session initiation information. If an RTSP message containing an SDP attachment is received, the *SDP* object is called and requested to fill the SDP structure. If there are unknown fields found in the SDP message and if there is an *SDPextension* object defined, then this object will be required to parse those extension fields.

### 2.2.2 The RTSP session handling part

Three main objects handle the RTSP sessions and the RTSP streams: *RTSPlib*, *RTSPsession* and *RTSPstream* (Fig. 2.2). The *RTSPlib* object manages RTSP sessions. It generates RTSP sessions when specifically requested by the application, or when the application uses the library in server mode and RTSP clients are connecting to the server.

An RTSP session is managed with the help of the *RTSPsession* object. This object handles all the RTSP messages that are using its connection, incoming or outgoing. The *RTSPsession* object generates streams (handled by *RTSPstream* objects) when requested by a **SETUP** RTSP method and destroys them when receiving a **TEARDOWN** RTSP method.

Incoming RTSP requests (usually on the server side) and incoming RTSP responses (usually on the client side) are handled using callback functions. These

callback functions are part of the *RTSPinterface* object and they are to be defined and implemented by the application. The reason for this is that it is for the application to decide what to do when an RTSP request or an answer to a previous request has come.

A data transport stream is managed with the help of the *RTSPstream* object. Each *RTSPstream* object keeps the RTSP state and transport parameters for its own stream.

For an application to be able to use the RTSP, RTP and SDP library, it has to initialize first the library in one of the three possible modes: client only, server only or both client and server mode. In the client-only mode the library is not listening for incoming connection, it can only generate RTSP session instances upon request by the application. In the server-only mode, the application can not specifically request RTSP sessions, it can only answer to RTSP requests of connected RTSP clients. In the third mode, the library can both listen and accept inbound RTSP connection and generate RTSP sessions on request. When creating an RTSP session upon request, the client application has to specify the URI of the requested resource. At initialization time, the application may specify an *RTSPinterface* object, an *RTSPextension* object and an *SDPextension* object. An application is quite useless if it does not specify an *RTSPinterface* object. This is because the application can not act as a server (because it can not handle incoming RTSP requests) and it can not act as a client either (because it can not handle the responses, and handling the responses of some commands like **DESCRIBE** and **SETUP** is critical).

Most of the RTSP processing inside the library is done inside the *RTSPsession* object. When a new session is created, an *RTSPsession* object is also created (instantiated) and this object receives an active connection. Each *RTSPsession* object with an active network connection is waiting to receive an RTSP message and to process it. When a new RTSP message arrives an *RTSPmsg* object is created to handle it. Inside this *RTSPmsg* object the message is parsed and the data structure defined by *RTSPmsg* is filled. The right session to handle this message has to be selected. This selection depends on the connection status, on the Session ID in the message and on the Session ID of the session (Algorithm 1 below). We need to make the session selection because a session can have a permanent connection or not, and at a certain point it can have a Session ID or not. If the client uses non-permanent connections, the server has to be able to identify the session and to take action within that session.

To facilitate the session selection, we have an array of sessions handled by the *RTSPlib* object. Only sessions with valid Session IDs are stored in this array. We are able to identify a session by knowing its Session ID. If a session connection is closed and if the session object (*RTSPsession*) does not have a valid Session ID, the session object (instance) is destroyed, because there is no way to reach that session object (instance) in the future. If it has a valid Session ID and the connection is closed, all data associated with that session are kept, because it is possible that the session is accessed in the future through a new connection. The complete algorithm for session handling and for deciding to which session an incoming message belongs is presented in Algorithm 1.

Each RTSP command is handled with the help of four functions: two functions

defined by *RTSPsession*, and two functions defined by *RTSPinterface*. The *RTSPsession* object defines a Request function and a Response function for each RTSP method. The *RTSPinterface* object defines an Indication and a Confirmation function for each RTSP method. These functions are called as follows (Fig. 2.3):

1. The client application wants to request an RTSP method (e.g. **PLAY**). The corresponding Request function (e.g. *PLAY\_Request*) is called with the desired parameters.

---

**Algorithm 1** Finding the proper RTSP session instance for a received message

---

- If the connection is closed, the session thread is going to be finished. If the session has no Session ID, the instance is destroyed.
  - If the received (current) message has a valid Session ID:
    - If the current session instance has a valid Session ID:
      - \* If they match, the message is processed in this session instance
      - \* If they do NOT match, it is an error and the current message and its connection are ignored
    - If the current session instance does NOT have a valid Session ID, then the Session ID from the current message is searched in the list of sessions:
      - \* If a session with the same Session ID is found:
        - If that session does NOT have a valid connection, the current connection is transferred there, the current message is processed in the found session and the current session instance is destroyed
        - If that session has a valid connection, it is an error and the current message and connection are going to be destroyed
      - \* If a session having the same Session ID as the current message was NOT found :
        - If the current session was supposed to receive a Session ID (e.g. answer to a setup request) then the current session is added to the list of sessions
        - If the current session was NOT supposed to receive a Session ID, an RTSP error is returned
  - If the current message does NOT have a valid ID:
    - If the current session instance has a valid Session ID, an RTSP error has occurred (no session present in the message)
    - If the current session instance does NOT have a valid Session ID, the current message is processed in the current session instance
-

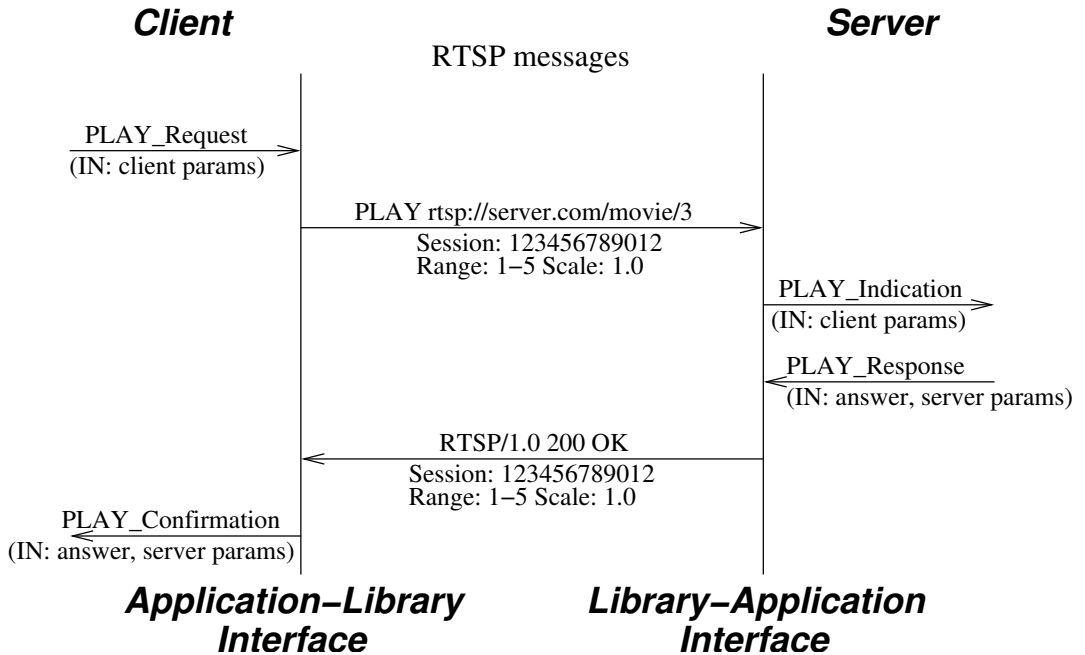


Figure 2.3: Functions handling RTSP methods. The interface with the application of the RTSP, RTP and SDP library.

2. The RTSP, SDP and RTP library on the server side receives the RTSP message from the client. The RTSP message is parsed, stored as an *RTSPmsg* structure and the proper *RTSPsession* object is identified using the Algorithm 1. The library then calls the Indication function corresponding to the received RTSP method (e.g. *PLAY\_Indication*). The definition of this function is provided by the server application.
3. The server application has processed the incoming request and it is calling the corresponding Response function (e.g. *PLAY\_Response*). Calling the Response function can be done from inside or outside the Indication function. The Response function is sending an answer back to the client.
4. The RTSP, SDP and RTP library on the client side receives the RTSP response message from the server. The RTSP response message is parsed, stored in an *RTSPmsg* structure and the proper *RTSPsession* object is identified using the Algorithm 1. The library then calls the Confirmation function corresponding to the received RTSP response method (e.g. *PLAY\_Confirmation*). The definition of this function is provided by the client application.

Several additional operations are performed when a **SETUP** request is involved. Before sending the **SETUP** request (on the client side) or after processing a successful RTSP **SETUP** request (on the server side), the library creates an *RTSPstream* object (instance) that will handle the data stream. If the client receives an error answer, the *RTSPstream* object is destroyed. There has to be an *RTSPstream* object (instance)

created on the client before sending the request, just in case the data stream will arrive faster than the **SETUP** answer.

For each data stream an *RTSPstream* object is created. An *RTP* object (instance) is associated with the *RTSPstream* object (instance) set up according to the corresponding **SETUP** Transport parameters. If the created data stream is an incoming stream, its corresponding network socket is polled for incoming data. When data is received, the RTP instance is responsible with data decapsulation. If the created data stream is an outgoing stream, the client application can send data using a function available in the library's API. Before the data is sent, the RTP instance is encapsulating the data in RTP packets.

### 2.2.3 Implementation and performance issues

The library was developed in C++ and C as a software package. It compiles on both Windows and Linux platforms due to an accompanying compatibility library. In this compatibility library we implemented some operating system specific primitives (networking, threads and thread synchronization related functions). The rest of the library calls the primitives from the compatibility library. At compiling time, the compatibility library detects the operating system and compiles the proper code, for that operating system. For example, the semaphore post and wait concepts are defined as `semaphore_post` and `semaphore_wait` functions inside the compatibility library. At compiling time these functions are defined to use `sem_post` and `sem_wait` if the detected operating system is Linux, or `ReleaseSemaphore` and `WaitForSingleObject` if the detected operating system is Windows.

The objects composing the RTSP, SDP and RTP library are implemented as C++ classes. The *RTSPinterface*, *RTSPextension* and *SDPextension* classes are purely virtual. The application using the library has to overload these C++ classes in order to use them. Each of the overloaded classes has a static member function that instantiates the class. The application provides the library with the overloading classes by supplying pointers to the instantiating functions when the library is initialized. If some of the classes are not needed (e.g. the application does not expect RTSP and SDP extensions), the application does not have to overload those classes and should pass to the library NULL pointers instead of instantiating functions.

The library is capable of processing hundreds of RTSP and SDP messages per second, on a PC class machine (with a 2 GHz CPU or faster). This figure is based on performance tests in which functions of the library were used to process an RTSP message containing all RTSP fields. In these tests the library functions were able to process around 2000 RTSP messages. We also gathered profiling data about the library when it was used in a real environment. The number of processed messages was then adjusted according to the profiling data: we multiplied the number of processed messages (2000) to the CPU percentage used by the parsing functions when the library was working in a real environment.

The processing of control messages is not likely to be the bottleneck in the implementation of an application based on this RTSP, SDP and RTP library. The most likely bottleneck when using the library would happen in streaming tens and hun-

dreds of Mbits/s using RTP. The reason for this is the encapsulation of data into RTP packets: to have place for the RTP header information, new memory has to be allocated, and the data to be sent has to be copied into this newly allocated memory. This process consumes lots of resources when streaming many tens of Mbits/s. A possible solution to this problem is to implement the RTP encapsulation in such a way that the data to be sent comes from the application with preallocated extra space required by the RTP header.

## **2.3 Applications of the library**

We used this library in three major applications, which are described in the next chapters:

### **2.3.1 Complete system for the transport of MPEG-4 content over the Internet**

In this application the aim was to design and implement a complete system for the transport of MPEG-4 content over the Internet [20, 80, 12]. The architecture of this system is described in details in Chapter 3.

The end-to-end system contains a MPEG-4 server and a MPEG-4 client using Delivery Multimedia Integration Framework (DMIF) [43] to communicate with each other. We replaced the DMIF Default Signalling with our library. The control part was implemented by mapping the DAI (DMIF Application Interface) primitives to a set of RTSP commands (SDP was also used). This alternative was preferred to DMIF Default Signalling since it allows interoperability with other RTSP-based media servers [9].

Using our library for this application required creating RTSP extension fields and also mapping the library's interface to DAI primitives as an additional library interface (see Fig. 2.1). The RTSP extension fields were necessary for carrying DMIF specific information that could not be mapped to standard RTSP fields.

The implementation of the architecture was successfully demonstrated at the 50th MPEG meeting in Maui, Hawaii, in December 1999. At that time, this architecture was one of the first two examples of a complete system for the transport of MPEG-4 content over the Internet based on mapping onto IETF protocols.

### **2.3.2 Plugin-based networked multimedia player for PC and STB**

This second application was aiming at creating a networked MPEG-2 player for Linux [81, 76, 86]. The architecture of this player is described in details in Chapter 4.

One of the main challenges in developing this player was the interoperability with a commercial video server, because the product manufacturer used its own extensions to RTSP and SDP within its server. The video server was not using RTP but a proprietary transport protocol, similar to RTP, described in the server documentation. We implemented this protocol in order for our library to be capable of understanding what the server is sending.

In this application we implemented an additional library interface, as in Fig. 2.1, because the video server was expecting the RTSP commands in a certain non-standard order. The server was not using the **DESCRIBE** command, but expected a **SETUP** command for the URI of an MPEG-2 transport stream [40, 52], that has both audio and video streams multiplexed. The answer to this **SETUP** command contains the SDP information that is usually sent as a response to a **DESCRIBE** command. If the **SETUP** command contained a special extension field, the server would start playing the stream immediately, without waiting for a **PLAY** command from the client.

The new mapping module hides from the application the complexity of interacting with this video server, instead providing it with few very simple commands, such as Open Stream, Play, Pause, End Stream.

### 2.3.3 Video on Demand (VoD) multimedia server

This application was developed in two distinct phases. The goal of the first phase was to create a VoD server in order to test the Networked Multimedia Player mentioned above. The server was designed and implemented on the top of the RTSP, RTP and SDP library as a light application, capable of streaming approximately 10 MPEG-2 Transport Streams. In this server application we are not using an additional library interface, because the application is simple enough and does not require such an additional interface.

In the second development phase the server evolved as a high-speed multimedia server capable of streaming up to 50 MPEG-2 streams (average of 4 Mbits/s per stream) using normal PC hardware and more than 100 MPEG-2 streams by using server-type PC architecture (66 MHz/64 bit PCI bus, RAID). The optimization of the VoD server in order to be able of such performance is described in details in Chapter 5.

## 2.4 Summary and author's contributions

In this chapter we addressed the interdependence of RTSP, RTP and SDP in the context of separating the delivery layer from the rest of the multimedia application. Extending RTSP and SDP was also considered. The solution proposed by the author to overcome these problems consists of an integrated RTSP, RTP and SDP library providing a common interface to all these three protocols. This library has been implemented and experimentally tested in several applications, proving its usefulness.

The scientific contribution of this approach has two main aspects:

1. We integrated RTSP, RTP and SDP protocols providing a single interface for them. This interface hides from the application unnecessary protocol details. This integrated package is a good starting point for hiding completely the delivery layer from the application.
2. Our integrated library allows RTSP and SDP protocols to be extended, as permitted by the standards. Extending these protocols is done through the

library interface, without modifying the library itself. The added value of this method is that one does not need to know the internal library details in order to extend it, and avoiding the modification of the library, the quality of its implementation is preserved.

The work presented in this chapter is based on [83], however, the library was also used in work previously presented in [20, 80, 12, 86, 81]. Work involving this library is also presented in [82, 76, 77, 84, 85].

The author of this thesis contributed to this work by designing the architecture of the library and by implementing its basic parts. Other people that helped with the implementation are Marius Vlad (the parsing functions and parts of the RTSP), Gonzalo Valencia (the initial implementation of the SDP) and Diana Calugarescu (initial implementation of the RTP).

## Chapter 3

# Transport of MPEG-4 over IP

The MPEG-4 standard [42] has its own solution for the remote retrieval of MPEG-4 content. This solution is called Delivery Multimedia Integration Framework (DMIF) and it is described in the part 6 of the MPEG-4 standard [43]. DMIF specifies its default signalling protocol, but it allows an application to use other remote retrieval protocols too, if it obeys the DMIF paradigm of separating the delivery layer from the application.

The problem of the DMIF Default Signalling (DDS) is that it is a new protocol and it is not interoperable with the existing, deployed systems.

In this chapter provide a solution to this problem by proposing the replacement of DMIF Default Signalling with IETF protocols: RTSP, RTP and SDP. To do this we map the DMIF primitives to these protocols, creating a complete, DMIF-based architecture for the transport of MPEG-4 content over the Internet using IETF protocols. This architecture preserves the DMIF concept of separation between application and delivery. In particular, we present an original mapping of MPEG-4 specific signalling primitives onto RTSP methods, which allows for interoperability between MPEG-4 clients and DMIF-aware or DMIF-unaware media servers.

### 3.1 Delivery in MPEG-4 standard

The MPEG-4 standard [42] deals with multimedia content composed of different media types involving multiple objects (also called streams). These objects can be of two types, media objects, which encode audio-visual information, and system objects, which encode the spatial and temporal information necessary for decoding and rendering the media objects. MPEG-4 supports many types of media: audio, video and still pictures, of both natural and synthetic origin. A media stream encoded with an MPEG-4 conformant encoder results in one or more objects that have a special internal structure and are generically called Elementary Streams (ES). The possibility of having several ES for a single media stream allows for encoding with several levels of quality.

MPEG-4 audiovisual scenes are composed of several media objects, organized in a hierarchical fashion (often called an MPEG-4 presentation). The system object that

specifies the composition and the rendering on output devices of the media streams is called Scene Description stream, or Binary Format for Scenes (BIFS) stream. A Scene Description stream associates time information to a geometrical hierarchy description of media objects, since the geometry may change in time.

The Object Descriptor (OD) system stream contains initialization information for any stream involved in an MPEG-4 presentation, including media streams, BIFS streams and other OD streams. A special OD stream called Initial Object Descriptor (IOD) contains initialization information about the streams that are involved in the MPEG-4 presentation at the very beginning.

The MPEG-4 standardized solution for Elementary Stream transport and control is called DMIF [43] and its purpose is to separate the delivery from the application and to ensure end-to-end signalling and transport interoperability between end-systems. Thus, the MPEG-4 application (player or server) becomes delivery unaware, and the transport becomes media unaware. The advantage for the application is that all the operations it performs on the media do not depend on how the media is transported, and the advantage for DMIF is that the transport operation does not depend on the media it carries (but it may depend on the QoS parameters associated with that media).

DMIF has a standard interface to the MPEG-4 application, for both control and media delivery. This interface is called DMIF Application Interface (DAI). Above DAI resides the Sync Layer (SL), which provides synchronization and packetisation for Elementary Streams that are passing through the DAI.

The DMIF layer consists of two separate component types: the DMIF filter and DMIF plugins. The role of the DMIF filter is two-folded: when the MPEG-4 client application (player) tries to open an MPEG-4 document, the filter has to choose the DMIF plugin that supports the transport protocol from the presentation's URI. Second, the filter acts like a buffer between the chosen plugin and the application, by forwarding commands and data.

An MPEG-4 application may have several DMIF plugins available (Fig. 3.1). These may include a DMIF plugin for retrieving MPEG-4 presentations from local storage and a DMIF plugin for retrieving presentations from a remote DMIF server. DMIF defines a network protocol to be used between the two DMIF instances, called DMIF Default Signalling.

In the next sections we describe the architecture and implementation of a DMIF plugin for remote retrieval based on standard IETF protocols: RTSP, SDP and RTP. For this purpose, we map the DMIF commands onto a subset of RTSP commands and the SL packets to RTP. The mapping allows for interoperability between a DMIF client using the RTSP/SDP/RTP DMIF plugin and an RTSP/SDP/RTP server, capable of streaming MPEG-4 presentations. The use of such DMIF plugin is in leveraging the current existing media servers and the current accumulated experience and achievements in implementing and running the above mentioned IETF protocols. It is believed that adding MPEG-4 streaming capabilities to an existing media server implementing RTSP, SDP and RTP is easier than adding a DMIF layer to the server, or even building a new one. Also, the implementation of the mapping between DMIF and RTSP/SDP/RTP described in this chapter is arguably simpler than implement-

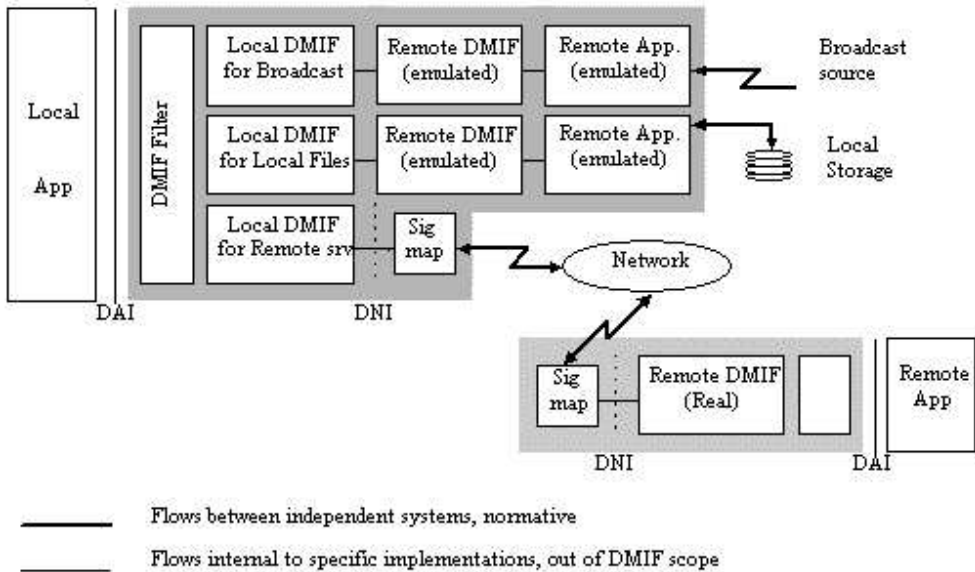


Figure 3.1: DMIF architecture and concept [30, 43].

ing the DMIF default signalling protocol.

The next sections describe also the implementation of two MPEG-4 client-server systems, sharing the same client. The server in one system is using the described DMIF plugin for streaming MPEG-4 client. The server in the second system is not DMIF-aware, it is only capable of streaming MPEG-4 content using RTSP, RTP and SDP. Building these two systems proves that the described mapping can be applied for both DMIF to DMIF and DMIF to RTSP/SDP/RTP communication.

## 3.2 Related work

Remote retrieval of MPEG-4 content has two main aspects: the control part and the transport part. The control part is responsible for the forwarding of the commands from a client to a server and for returning a response. The transport part is responsible for the delivery of the data streams to the client. There are several protocol choices for both the control part and the transport part. While highly dependent on each other, the control part and the transport part can be addressed separately, and any control protocol can be combined with (almost) any transport protocol.

Control protocols used for MPEG-4 retrieval can be split into several categories, as follows:

1. MPEG-4 control using DMIF framework:
  - (a) DMIF Default Signalling
  - (b) RTSP

- (c) Session Initiation Protocol (SIP) [51]
  - (d) Session Announcement Protocol (SAP) [50]
2. MPEG-4 control using IETF protocols outside DMIF framework:
- (a) RTSP
- Transport protocols used for MPEG-4 retrieval can also be split into several, similar categories:
3. MPEG-4 transport using DMIF framework:
- (a) DMIF default transport protocol
  - (b) RTP
4. MPEG-4 transport using IETF protocols outside DMIF framework:
- (a) RTP

The work presented in this chapter deals with remote retrieval control using RTSP inside DMIF framework (1.b), mainly targeted at unicast streams and having Video on Demand (VoD) as main application. Similar work, aiming to control MPEG-4 using RTSP is described in [134], but the presented system targets only real-time multicast streaming of a live presentation. Implementation work involving DMIF signalling over RTSP is reported in [11], but no details about the mapping are given. Control related issues involving DMIF and RTSP have been the subject of discussions between MPEG System Group and Internet Engineering Task Force (IETF). Three Internet Drafts have resulted from this effort [8, 24, 121]. In [8] the author proposes a standardization-based migration from DMIF Default Signalling to a DMIF signalling containing RTSP elements. The starting point for this transition is envisioned as the carriage of RTSP-specific elements inside User-to-User data of DMIF signalling commands. In [24] it is proposed to use RTSP elements (commands and fields) for carrying (mostly) VoD specific commands over DMIF. This is motivated by the fact that DMIF does not specify any stream command such as **PAUSE** or **PLAY**, but allows the application layer to have its own protocol. The architecture and implementation aspects of a system using [24] are presented in [65, 14]. In [121] guidelines for the carriage of MPEG-4 content over IETF protocols (RTP, SDP and RTSP) are proposed. While DMIF is not mentioned at all in this document, the proposed framework can be applied for both DMIF-based and non-DMIF-based delivery of MPEG-4 over IETF protocols. The work presented in this thesis follows these guidelines, except for the guidelines in paragraphs 3.1 and 5.2 of [121], both paragraphs referring to the way of delivering the IOD to the client. The method proposed in [121] consists in specifying a location for the IOD in the SDP attached to a **DESCRIBE** command, and in retrieving the IOD with the second issued **DESCRIBE**. The method proposed in this chapter embeds the IOD in the SDP by encoding it using the base64 algorithm [61].

Work involving streaming MPEG-4 using RTSP as control protocol, but not using DMIF or any other MPEG-4 system features (2.a), is presented in [110, 140].

Work aimed at controlling MPEG-4 using SIP (1.c) is described in [5], where the authors present a videoconferencing system based on MPEG-4.

In [94] it is described a MPEG-4 ES broadcasting system using DMIF for the transport part (3.a) and SAP for the control part (1.d).

Work involving streaming in the DMIF context but using DMIF Default Signalling (1.a) is described in [102, 47, 48, 4, 6, 3, 58, 64, 14, 100]. In [102] and [47] the authors present two streaming scenarios (broadcast and remote retrieval) using their own DMIF filter for IM1 (IMplementation 1, [41]) and own server implementations. In [48], the same system is enhanced and advanced multicast capabilities are added. In [6] and [3] a VoD system using QoS over DiffServ is presented. More details about the algorithm for marking the video packets for DiffServ can be found in [4]. In [58] it is presented one of the first commercially available DMIF implementations using Default Signalling and also QoS. A complete system based on [58], using IM1 as the MPEG-4 player is presented in [64]. Another system using DMIF Default Signalling for the control part, but using RTSP commands inside User to User data (as described in [24]) and RTP for the transport part is described in [14, 65]. In [100] a DMIF based application supporting dynamic QoS management based on RSVP is presented.

The MPEG-4 media transport using RTP has been addressed in a joint effort between IETF and MPEG System Group, resulting in the publication of a standard [66] and several Internet Drafts [10, 130, 45, 46, 133]. These documents address the definition of payload formats for the transport of MPEG-4 content over Internet. In [66] it is proposed a packetisation scheme for audio and video Elementary Streams, also handling fragmentation, but the proposed scheme does not take into consideration the MPEG-4 Systems. In [10] a SL packetisation scheme that carries one SL packet in one RTP packet is proposed. The packetisation scheme splits the SL packet header in 3 parts: information carried by the RTP packet header, general information, carried in the payload header and MPEG-4 system specific information carried as remaining SL header. This packetisation scheme maintains compatibility with [66] and is also presented in [9, 11, 12]. In [130] it is defined an RTP payload format that specifies the carriage of SL information. In [133] an RTP packetisation for the H.264 video codec is presented. Work describing the transport of MPEG-4 streams over RTP also makes the subject of several articles, from which [4, 97] contain good reviews of work in this area.

The DMIF plugin described in this chapter implements the RTP payload format for MPEG-4 streams described in [10]. The described mapping between DMIF and RTSP is based on experience gained from previous work [80, 20, 65].

### 3.3 Short presentation of DMIF and DAI

The DMIF Application Interface (DAI) is a set of generic primitives that separates the DMIF Application from transport specific mechanisms (DAI can also be considered the API of DMIF). The existence of DAI makes the delivery technologies to be

completely transparent and abstracted, so a new delivery technology may be added without altering the primitives of the DAI or the application using it. MPEG-4 Part 6 [43, 30] (DMIF) does not impose any programming language or syntax for DAI. Each DMIF implementation has to define its own DAI according to DMIF semantics outlined in [43]. Syntactically, each DAI implementation can be different, depending on the programming language used (e.g. C, C++, Java) and on the style of the programmer.

The basic concept of the DMIF paradigm is the transport of streams. A stream may contain one MPEG-4 Elementary Stream (ES) or more, multiplexed, Elementary Streams. Streams coming from the same source are grouped and form one session. MPEG-4 application may have many streams from many sources, thus there is no mandatory link between an MPEG-4 application and a DMIF session. However, most of the times an MPEG-4 application resides in one place and thus only one DMIF session is necessary for the application retrieval.

After a DMIF application has opened a session it may request some streams. Each stream transport mechanism is called Channel in DMIF terminology. The DMIF application may open a session to another DMIF application or just perform some operations (e.g. open one or more files) on the local system. After requesting some streams (channels), the DMIF application may issue some commands for each opened channel. We can assume that most of these commands would be streaming and QoS related, however, DMIF does not define any such command specifically. It is for the applications to define the commands they need and solve any interoperability issues that may arise. One can imagine many such commands, ranging from “Play”, “Pause”, “Fast Forward”, “Fast Reverse” and their associated data (e.g. the speed for Fast Forward and Fast Reverse), in a VoD like MPEG-4 application, to “Move the monsters faster”, “I changed my weapon type to weapon\_type\_3” in a Game MPEG-4 application, or “I am buying the product advertised now on my TV screen, here is my credit card information” in an interactive TV broadcast MPEG-4 application scenario that many IT companies would so much like to see happening.

For the transport of multiple Elementary Streams in the same Channel, DMIF defines two multiplexing layers: Flexible Multiplexing - FlexMux and Transport Multiplexing - TransMux, (see Fig. 3.2). The first multiplexing layer, FlexMux, is syntactically defined by DMIF and allows grouping of Elementary Streams with a low multiplexing overhead. Multiplexing at this layer may be used, for example, to group Elementary Streams with similar QoS requirements, reduce the number of network connections or the end-to-end delay.

The second multiplexing layer, TransMux, is network dependent and so, MPEG-4 only specifies the interface to this layer. Concrete mapping of the data packets and control signaling must be done in collaboration with the bodies that have jurisdiction over the respective transport protocol. Any suitable existing transport protocol stack such as RTP/UDP/IP, AAL5/ATM, or MPEG-2’s Transport Stream over a suitable link layer may become a specific TransMux instance. The choice is left to the end user/service provider, and allows MPEG-4 to be used in a wide variety of operation environments. A good example of a TransMux would be the grouping of several audio Elementary Streams that are sent using RTP/UDP into a single network stream

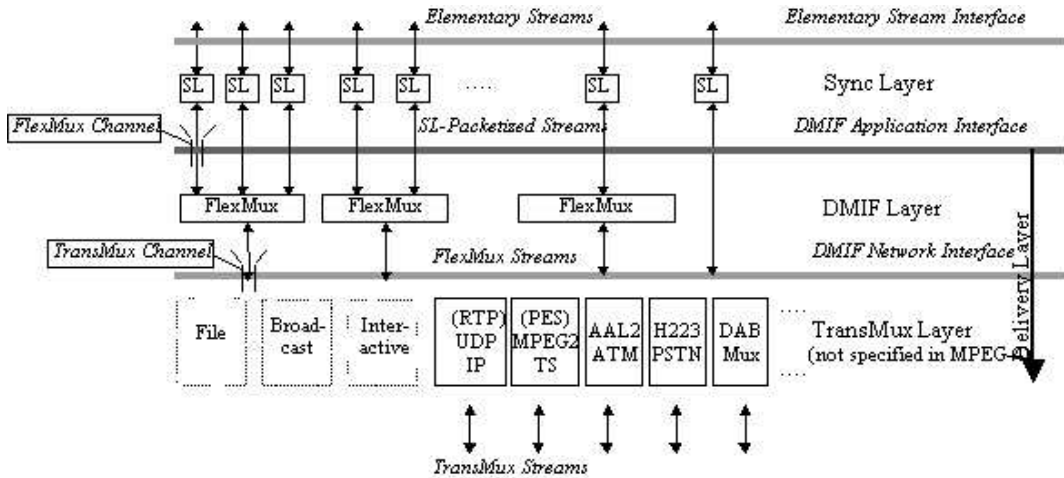


Figure 3.2: FlexMux and TransMux Channels in DMIF.

(pair of sender and receiver IP and ports). The receiving DMIF would be able to demultiplex the Elementary Streams based on the Synchronization Source Identifier (SSRC) field of incoming RTP packets. Similar to FlexMux would be to multiplex packets from all the audio streams into one RTP payload and send it as one RTP stream.

We can summarize the primitives associated with DAI into 4 categories:

1. Service primitives: *DAI\_ServiceAttach* and *DAI\_ServiceDetach*. These primitives allow the application to create and destroy a (single) DMIF session.
2. Channel management primitives: *DAI\_ChannelAdd* and *DAI\_ChannelDelete*. These primitives allow the application to create and destroy DMIF channels. Each of the two primitives may handle several DMIF channels.
3. Channel control primitives: *DAI\_UserCommandAck* and *DAI\_UserCommand*. They provide the application with a mechanism to issue acknowledged or non-acknowledged commands associated with channels.
4. Data primitives: *DAI\_SendData* and *DAI\_OnDataReceived*. These primitives allow the application to send and receive data over the created DMIF channels.

Each primitive in the first three categories has four API functions associated with it: Request, Indication, Response and Confirm, abbreviated as Req, Ind, Rsp and Cnf. The Request and Response are request-type functions and Indication and Confirm are callbacks (similar as the interface functions for the RTSP, SDP and RTP library described in Chapter 2). In a client-server DMIF scenario, in order to setup the session, the client calls the *DAI\_ServiceAttach\_Req* function. The DMIF instance on the client contacts the peer DMIF instance on the server side. Here the *DAI\_ServiceAttach\_Ind* function is called by the DMIF in order to let the server MPEG-4 application know about the incoming connection. The body of the

*DAI\_ServiceAttach\_Ind* function is not a part of DMIF, but it is provided by, and it is a part of the MPEG-4 application. After the server has done processing the incoming session creation request, it calls the response function, *DAI\_ServiceAttach\_Rsp*. The place of the calling is implementation dependent and may happen from inside the *DAI\_ServiceAttach\_Ind* or from outside this function, if asynchronous processing is involved. When the response reaches back the client, the DMIF layer calls *DAI\_ServiceAttach\_Cnf* callback function in order to let the application know about the response issued by the peer DMIF. Similarly as the *DAI\_ServiceAttach\_Ind* function, the body of the *DAI\_ServiceAttach\_Cnf* is not part of DMIF, but it is provided by and it is part of the MPEG-4 application.

The rest of the DAI primitives in the first three categories have similar calling scenarios, except for the *DAI\_UserCommand* primitive that does not have an answer, thus its Rsp and Cnf functions are missing. One should note that for an MPEG-4 application using DMIF which acts as a client only, it is not necessary to implement the Ind functions. Similarly, for an MPEG-4 application that expects to act as a server only, it is not necessary to implement the Cnf functions.

### 3.4 Mapping of DAI primitives onto RTSP methods

This section presents the technical details for the usage of RTSP as control protocol for an IP unicast based DMIF plugin. The transport protocol is supposed to be RTP, so that RTSP and RTP are replacing the DMIF default signalling protocol. This mapping does not cover the transport part, so no further details concerning the transport protocol are covered here.

The section is structured in 4 parts:

1. Important issues appearing when mapping DMIF to RTSP
2. Mapping of DMIF commands to RTSP, on the client side. This part addresses the issue of creating the control part for a DMIF client plugin that is able to contact a standard RTSP server.
3. Mapping of RTSP commands to DMIF, on the server side. This part addresses the issue of creating the control part for a DMIF server plugin that is able to understand and translate to DMIF commands the requests of a normal RTSP client.
4. Interoperability and optimization issues for DMIF clients and server that uses RTSP for signalling. This part addresses the issue of creating better communication between a DMIF client and a DMIF server that both use RTSP as control protocol.

The Annex I, "DMIF Application Programming Interface - Syntax Definition" of the document ISO/IEC JTC 1/SC 29/WG 11 N3060 was used to provide information about DMIF syntax and parameters for this mapping.

### 3.4.1 Important issues concerning mapping DMIF to RTSP

#### 3.4.1.1 Translating between ES\_IDs, DMIF Channel IDs and RTSP URIs

The MPEG-4 Sync Layer (SL), DMIF and RTSP have different naming conventions and different ways to identify MPEG-4 streams:

- For the Sync Layer an MPEG-4 stream is viewed as Elementary Stream and is identified by its Elementary Stream ID (ES\_ID).
- For the DMIF layer an MPEG-4 stream is viewed as a channel and is identified by its Channel ID (ch\_id).
- For the RTSP layer a stream (MPEG-4 in our case) is viewed as a stream and is identified by its URI.

When the MPEG-4 client application requests channels it will provide the DMIF layer with the ES\_ID of the requested channels. It is DMIF's responsibility to generate Channel IDs associated with ES\_IDs (they can be the same). The MPEG-4 application (Sync Layer) will receive the DMIF Channel IDs associated with the requested ES\_IDs in the *DAI\_ChannelAdd\_Ind* function (for the server) and *DAI\_ChannelAdd\_Cnf* function (for the client).

Associating RTSP URIs with DMIF Channel IDs has to be done inside the mapping of DMIF to RTSP. We see several ways of constructing the RTSP URI to carry the channel information:

- The channel ID is the last element of the URI path:

```
rtsp://server/linda.mp4/1
```

The advantage of this method is its simplicity. Similar URI forms may include:

```
rtsp://server/linda.mp4/esid1
```

```
rtsp://server/linda.mp4/esid1
```

- The channel ID is carried as data inside the URI query part (as specified by URI standard [13], similar to HTTP URL when using the GET method [29]):

```
rtsp://server/linda.mp4?ch_id=1
```

The advantage of this method is that multiple channels may be specified in one URI:

```
rtsp://server/linda.mp4?ch_id=1&ch_id=2&ch_id=4
```

The disadvantage of this method is that URIs containing queries are not yet standardized for RTSP (present in the latest RTSP Internet Draft [120], but not in the RTSP RFC [118]).

- The association between RTSP URI and the ES\_ID is done using SDP attributes:

```
a=control:rtsp://localhost/linda.mp4/some_id
```

```
a=x-mp4-esid:2113
```

This method has one important disadvantage: it relies on SDP for making the association, so all the streams in the session must be described in the SDP attachment. Depending on design and implementation choices, this may be not always possible (see case 3.b from the next subsection). The method is also somehow redundant, because the information contained in the stream control URI should be enough to identify an ES\_ID, however the method needs additional information to make the association.

### 3.4.1.2 RTSP/SDP require information about all streams

One major difference between DMIF and RTSP paradigms is that DMIF does not require information about all streams at the beginning of a session, but the RTSP paradigm requires so, in various forms, e.g. by SDP attachment. In a MPEG-4 application using DMIF scenario, after retrieving the IOD using **ServiceAttach**, the MPEG-4 client may discover only two streams referred inside the IOD: the BIFS and OD streams of the MPEG-4 presentation. The OD stream may, after some time, refer to other, possibly media, streams. DMIF should retrieve the new streams without any problem, even if their existence was not known to DMIF at the beginning of the session. Because of this reason and because we need to have an SDP describing any MPEG-4 presentation streamed with RTSP, to be in line with the RTSP paradigm we need to answer two questions:

- By whom, when and how is the SDP created.
- What information is contained in the SDP.

There can be several scenarios for the creation of SDP, each scenario answering to both questions above:

1. The SDP is created offline, together with the MPEG-4 presentation, and stored on a disk. When a client opens a session, the SDP is either passed to DMIF or there is a way in which DMIF is able to retrieve the SDP using the presentation's IOD.

The advantage of this method is that it is relatively simple to implement. The disadvantages are that it is not clean (it requires an extra module to generate the SDP offline) and the usage of such a server may become complicated. A new MPEG-4 presentation has to be “registered” with DMIF and RTSP in order to get an SDP and associate that information with the IOD of the presentation.

2. The SDP is created by the MPEG-4 server application and passed to DMIF along with the IOD as a parameter of the *DAI\_ServiceAttach\_Rsp* function.

This method is also not clean, because the server needs to have knowledge about delivery layer above DAI. This undermines the whole DMIF philosophy.

3. The SDP is created “on the fly” from the IOD, inside the *DAI\_ServiceAttach\_Rsp* function. There are two possible sub-scenarios:
  - (a) The IOD contains information about all the streams that make the presentation. This is perfectly possible for VoD-like applications, for which the use of RTSP signalling inside DMIF is targeted. However, having such a constraint upon the IOD would also put constraints upon the creation environment of such a MPEG-4 application.
  - (b) The IOD only contains information about several streams from the presentation. In this case it is up to the RTSP layer/ implementation to request the right streams (create the right URI) with only the ES\_ID and the RTSP session URI, and no other information usually carried inside SDP. One consequence of not having the SDP information is that the RTP payload type must be known to both server and client.
4. The SDP is created “on the fly” inside the *DAI\_ServiceAttach\_Rsp* function from additional information available to the DMIF layer. This method may be regarded as not clean, since it is very implementation-dependent how the additional information is passed to DMIF, what this additional information contains (does it still keep DMIF media-unaware), and how the SDP is generated from this information.

#### 3.4.1.3 Synchronization for one-to-many command mapping

One of the major differences between DMIF and RTSP is that in DMIF a command (**ChannelAdd**, **UserCommand**, **ChannelDelete**) may be specified to an arbitrary set of channels/streams. In RTSP each stream/channel has to be created by a separate **SETUP** command and a **PLAY** or **PAUSE** command can be issued to a single URI. Usually this URI controls either the whole session or an individual stream. So we can either **PLAY/PAUSE** all the streams in an RTSP session at the same time, or one stream at a time.

If a DMIF **UserCommand** is issued to several Channels, the trivial approach would be to issue a separate RTSP command for each stream. However there are two problems associated with this approach:

1. The total command delay is at least  $N \cdot \text{RTD}$  (Round Trip Delay) of network packets between client and server, where  $N$  is the number of channels to which the command is issued.
2. De-synchronizations may appear due to issuing the commands at different times, for streams with the same time base (streams that need to be synchronized with one another) and for MPEG-4 clients/receivers with insufficient receiving buffers.

There are several solutions that solve the above mentioned two problems (some of the ideas presented in this sub-section belong to Guido Franceschini, Andrea Basso and David Singer from the MPEG committee):

1. Create a separate RTSP session for each group of streams with the same time base in a DMIF session. This solution would allow to send a single RTSP command to the session control URI and modify the status of all the streams in that session (having the same time base). For example, if a session has 9 streams and first 3 have a time basis, the next 3 have another time basis and the last 3 have another one, the client may create 3 RTSP sessions. If so, each of these sessions should have a control URI (all sessions can have the same control URI). A **PLAY** RTSP command issued to the control URI and having the second session as the RTSP Session ID, will have effect on all 3 streams in the second time base group, making them play simultaneously.

However, this solution is not able to handle gracefully all situations. For example, if we have a video with two associated audio streams (one coding the background sounds, the other two people talking in the foreground) and all three streams have the same time base, with this solution we can not pause/mute all the audio streams simultaneously. Additional problems appear when we have to issue commands to non-void intersecting sets. Also closing the DMIF session requires closing several RTSP sessions, thus requiring several **TEARDOWN** commands.

2. Create one RTSP session corresponding to the DMIF session and define several control URIs in the associated SDP for different stream sets. To issue a command to a predefined set, we only have to issue the command to the set's control URI as defined by the SDP.

While this solution can solve the problem of non-void intersecting sets and also the problem of issuing several commands when closing the DMIF session, one can argue that it is difficult to specify at session initialization time all the possible sets of streams for which commands will be issued simultaneously. In addition, there is no standard SDP method of defining a control URI for several streams.

3. Create one RTSP session corresponding to the DMIF session and agree on a convention to specify several streams with a single URI. This would allow a client to send a single RTSP command to several streams, by specifying them in the URI. This is our preferred solution and it is described in details in Section 3.4.6.3.

While this solution potentially solves both problems mentioned before, the RTSP client and server must share the same convention of specifying several channels/streams in a single URI.

### 3.4.2 Mapping of DMIF commands to RTSP, on the client side

This mapping assumes that the server is an RTSP/RTP server capable of streaming MPEG-4 content, not necessarily DMIF-aware. The client is an MPEG-4 terminal that has a DMIF filter and loads a DMIF plugin to communicate with the RTSP/RTP server. The mapping describes how this plugin should handle DMIF commands and how to translate them into RTSP commands.

#### 3.4.2.1 Service initialization: ServiceAttach

For the mapping of the **ServiceAttach** request, we have two cases:

1. The client already knows the IOD and all the necessary initialization information.
2. The client does not know the IOD and/or other necessary initialization information

For the first case, when all the initialization information is already known, there is no need to send any RTSP request.

For the second case, the service initialization and IOD retrieval is to be done using one of the 3 proposed methods:

1. Two RTSP **DESCRIBE** commands. The first **DESCRIBE** command retrieves the SDP information, the second **DESCRIBE** command retrieves the IOD. The disadvantage of this method is that the final answer comes after two round-trips delay. This method is also proposed in [121].
2. Content-Type Multipart/Related SDP, IOD. The advantage of this method is that the delay is only one round-trip. If the RTSP is carried over TCP, this will also handle the fragmentation in the case the resulting RTSP packet is too big to be carried over one IP packet. The disadvantage is that the client should accept multipart related attachments. Example of such SDP attachment:

```
C->S: DESCRIBE rtsp://server/mp4pres RTSP/1.0
S->C: RTSP/1.0 200 OK
      Content-Type: Multipart/related;boundary=IODdata
      --IODdata
      Content-Type:application/sdp
      a=control:rtsp://server/linda.mp4
      m=application 0 RTP/AVP 96
      a=rtpmap:96 X-MP4-ES/1000
      a=control:rtsp://server/linda.mp4/esid1
      a=x-mp4-esid:1
      --IODdata
      Content-Type:Application/octet-stream
```

```

Content-Description: IOD data
Content-Transfer-Encoding: base64
AoCAgHIAHwEBAgEBA4CAgCsAAAAF38=
--IODdata--

```

In the example above the first multipart attachment is the SDP description of the session (bounded by the first and second IODdata tag). The first line of the SDP description (`a=control:rtsp://server/linda.mp4`) concerns the MPEG-4 session as a whole and provides the control URL of the session. The next four lines concern one media stream belonging to the session. The first media line (`m=application 0 RTP/AVP 96`) describes the MIME media type, how this media can be sent by the server and its MIME payload number. We find from this SDP line that the media is application (unspecified), sent by the server using RTP, Audio-Video Profile, and the MIME payload number is 96 (a custom payload). More information about this custom payload can be found in the second SDP media line (`a=rtptime:96 X-MP4-ES/1000`). We find here the name of the custom payload (X-MP4-ES) and the frequency of the clock used to generate the RTP timestamps in the RTP stream carrying this custom payload (1000 Hz). The third SDP media line (`a=control:rtsp://server/linda.mp4/esid1`) provides the control URL for this media, to be used in such commands as **PLAY** or **PAUSE** concerning this media stream. The fourth SDP media line (`a=x-mp4-esid:1`) provides an attribute concerning this media stream. The name of the attribute is x-mp4-esid and its value is 1. For the peer MPEG-4 client, the meaning of this attribute is the ES\_ID of the media stream.

The second multipart attachment in the example above (bounded by the second and third IODdata tag) is the IOD, encoded using the base64 algorithm.

3. One **DESCRIBE** command carrying both the SDP information and the IOD, encapsulated in the SDP (base64 encoding) as a session attribute. This method also limits the delay to one round-trip. This is our preferred solution and the one described in the implementation section (3.5). When the author presented this idea to the scientific community, it was argued that the method requires some modifications to the SDP standard [49], namely modification of the maximum byte-string length and the allowed bytes in byte-string. However, the author did not find any reference to any limitation of a byte-string in the SDP standard. Below we present an example of such SDP attachment:

```

S->C: RTSP/1.0 200 OK
Content-Type:application/sdp
Content-Length: 663
a=control:rtsp://server/linda.mp4
a=X-MPEG4-IOD:AoCAgHIAHwEBAgEBA4CAgCsAAAAF38=
m=video 0 RTP/AVP 96
a=rtptime:96 mpeg4-s1/1000

```

```

a=control:rtsp://server/linda.mp4?streamid=2115
a=mpeg4-esid:2115

```

The SDP part of the example above is very similar with the previous SDP example. There is an extra line (), at the session level, that carries the value of the IOD as the value of an attribute (named X-MPEG4-IOD). The IOD is encoded using the base64 algorithm.

The SDP in the example above contains one media, described by four SDP media lines. The first media line (`m=video 0 RTP/AVP 96`) identifies this media as being a video stream, sent by the server using RTP with the Audio-Video Profile and having the MIME payload number 96 (a custom payload number). The second media line (`a=rtpmap:96 mpeg4-sl/1000`) identifies the payload number 96 as being an mpeg4-sl, and the clock used to generate the RTP timestamps for this media stream has the frequency equal with 1000 Hz. The third media line (`a=control:rtsp://server/linda.mp4?streamid=2115`) identifies the control URL for this media stream. The fourth media line (`a=mpeg4-esid:2115`) provides the ES\_ID value for this media stream.

A complete command mapping (both client and server) is presented in Fig. 3.3.

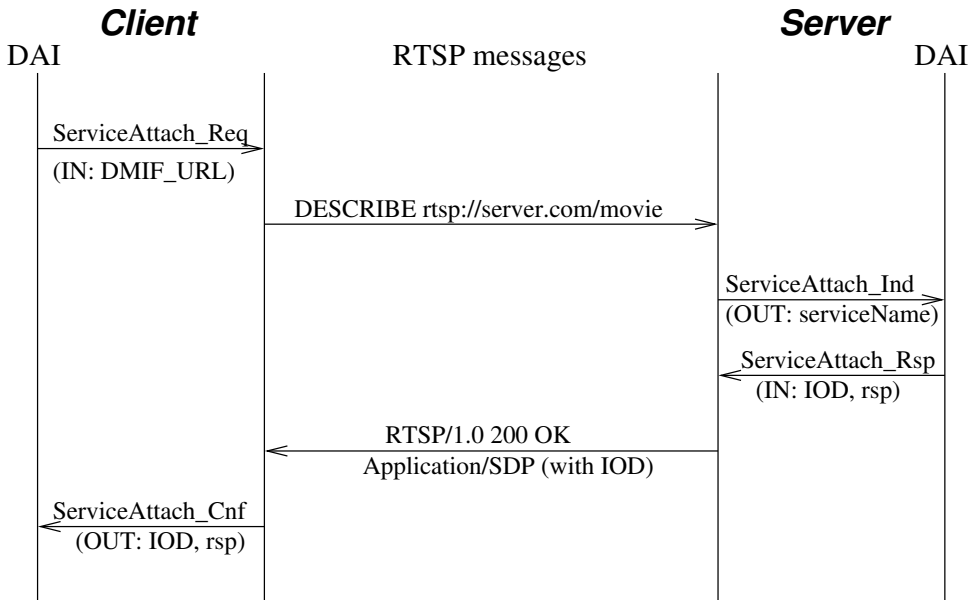


Figure 3.3: Mapping of ServiceAttach.

Parameters mapping: The usage of DMIF parameters when using RTSP should be as follows:

For the `DAI_ServiceAttach_Req` function, that has the following parameters:

- `TEMPORARY_ID_HANDLE` - to be used internally by DMIF. If the DMIF instance supports concurrent commands, this parameter can be mapped to a (RTSP Session ID, CSeq RTSP field) pair.

- URI - should be the URI for the RTSP **DESCRIBE** command(s).
- The user-to-user data (if any) can be send to the RTSP server by attaching it to the (first) **DESCRIBE** request of the method used (one of the three described above). If the client already knows all the necessary initialization information, a **DESCRIBE** request containing the data as an attachment may be send.

For the *DAI\_ServiceAttach\_Cnf* function, that has the following parameters:

- **TEMPORARY\_ID\_HANDLE** - the handle that was passed to the corresponding *DAI\_ServiceAttach\_Req* function. If the implementation relies on RTSP for mapping this parameter, it can be generated from the (RTSP Session ID, Cseq RTSP field of the response) tuple.
- **SESSION\_ID\_HANDLE** - a Session ID generated by DMIF (can be a pointer to the session instance in a C++ implementation or a pointer to the session data structure in a C implementation)
- **RESPONSE\_CODE** - a response mapped from the RTSP response. The response mapping is described in Section 3.4.5.
- The user-to-user data should point to the IOD that was retrieved either using one of the three methods proposed above, or known a priori.

### 3.4.2.2 Opening channels: ChannelAdd

A channel is opened using the **SETUP** RTSP command. A **ChannelAdd\_Req** command can have more than one channel as input parameter. For setting up each of these channels, a separate **SETUP** command has to be issued for each channel. A complete command mapping (both client and server) is presented in Fig. 3.4.

Parameters mapping: The usage of DMIF parameters when using RTSP should be as follows:

For the *DAI\_ChannelAdd\_Req* function, that has the following parameters:

- **SESSION\_ID\_HANDLE** - the DMIF Session ID, returned by the *DAI\_ServiceAttach\_Cnf* function
- For each channel in the *DAI\_ChannelAddReqLoop*:
  - **TEMPORARY\_ID\_HANDLE** - to be used internally by DMIF. If the DMIF instance supports concurrent commands, this parameter can be mapped to a (RTSP Session ID, Cseq RTSP field) tuple.
  - *DAI\_ChannelDescriptor* - contains QoS parameters. These parameters cannot be mapped to any RTSP command, field or attribute. Hints about their usage are given in Section 3.4.4

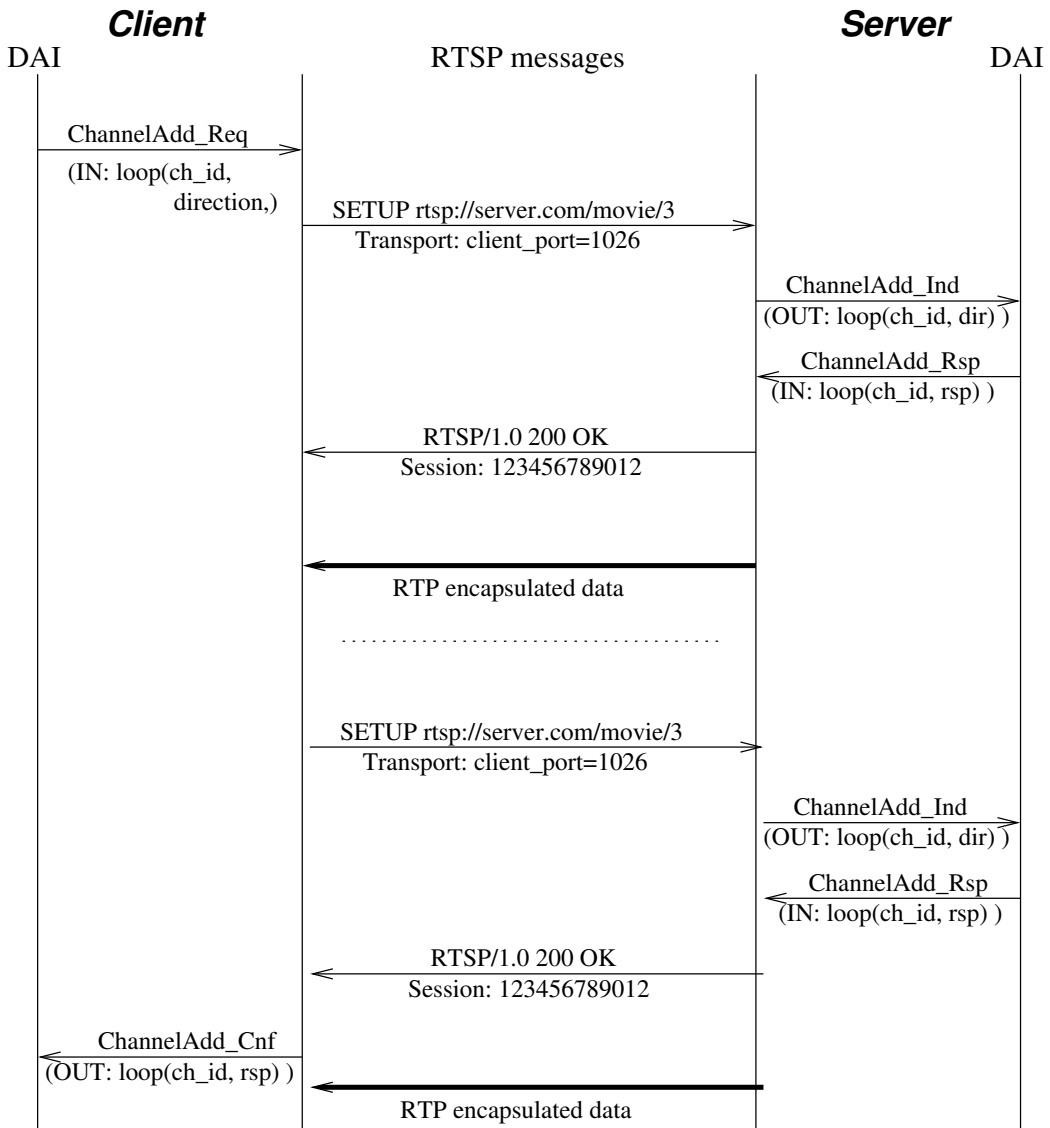


Figure 3.4: Mapping of ChannelAdd.

- CHANNEL\_DIRECTION - DMIF specifies three directions: upstream, downstream, bi-directional and unspecified. The direction can be mapped to the "mode" parameter of the Transport field of a **SETUP** RTSP command. A downstream direction is mapped to "mode=play", an upstream direction is mapped to "mode=record" and a bi-directional stream is specified as "mode=play, record". The unspecified direction can be considered as being downstream, by default.
- User-to-user information - this information should contain the ES\_ID of the requested stream and it is the only way of identifying which of the streams of the session the application is requesting. The MPEG-4 appli-

cation identifies a stream by its `ES_ID`. RTSP identifies a stream by its URI. DMIF identifies a stream (channel) by its Channel ID (`ch_id`). A method of making the correspondence between the URI of the stream and the `ES_ID` is described in [121]. The general idea is to have a SDP custom attribute for each stream that should specify the stream's `ES_ID`. For each channel, DMIF will generate the corresponding channel ID (can be a pointer to the channel instance in a C++ implementation or a pointer to the channel data structure in a C implementation).

For the `DAI_ChannelAdd_Cnf` function, that has the following parameters:

- `SESSION_ID_HANDLE` - the DMIF Session ID, returned by the `DAI_ServiceAttach_Cnf` function
- For each channel in the `DAI_ChannelAddCnfLoop`:
  - `TEMPORARY_ID_HANDLE` - the handle that was passed to the corresponding `DAI_ChannelAdd_Req` function. If the implementation relies on RTSP for mapping this parameter, it can be generated from the (RTSP Session ID, Cseq RTSP field of the response) tuple.
  - `RESPONSE_CODE` - a response mapped from the RTSP response. The response mapping is described in Section 3.4.5.
  - `CHANNEL_ID_HANDLE` - a DMIF generated ID for the channel. Subsequent references to this channel ID will refer to this stream. For this reason, the DMIF plugin needs to internally keep a correspondence between channel IDs and RTSP URIs.
  - User-to-user information - if the **SETUP** response contained an attachment, this is passed to the client using this parameter.

### 3.4.2.3 Controlling the channels: UserCommand

After the client application has opened channels, it can issue commands on those channels.

DMIF only defines a generic user command, since there cannot be a priori knowledge about specific commands needed by an application. Because we do not aim at complex interactive commands, we only support **PLAY** and **PAUSE** as user commands. An array (loop) of user commands is provided to the `DAI_UserCommand_Req` function.

These two commands can be mapped over the **PLAY** and **PAUSE** RTSP commands. The **PLAY** command can also carry a Range and a Speed, as RTSP fields. The pause command can only carry a Range.

We can issue **PLAY** or **PAUSE** to one stream, to all the streams having the same RTSP Session ID or to arbitrary streams in an RTSP session by specifying an URI as described in Section 3.4.6.3. A complete command mapping (both client and server) is presented in Fig. 3.5.

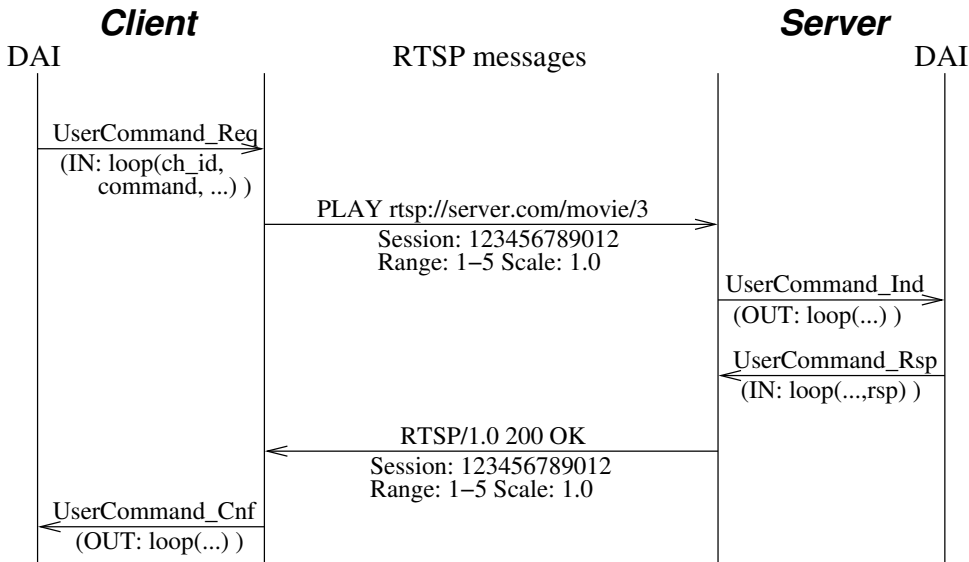


Figure 3.5: Mapping of UserCommand.

Parameters mapping: The usage of DMIF parameters when using RTSP should be as follows:

For the *DAI\_User\_CommandReq* and *DAI\_UserCommandAck\_Req* functions, which have the following parameters:

- **TEMPORARY\_ID\_HANDLE** - temporary handle for the command, used internally by DMIF. If the DMIF instance supports concurrent commands, this parameter can be mapped to a (RTSP Session ID, Cseq RTSP field) tuple.
- User-to-user data - this data contains the actual command and its parameters. For an RTSP plugin this data may contain:
  - One of the 2 supported commands, **PLAY** or **PAUSE**. Each of these commands will map directly to the corresponding RTSP command.
  - A playing position or range, or pausing position. The position or range can map to the Range RTSP field.
  - A playing speed (for **PLAY** commands only). This can map to the Speed RTSP field.
- For each channel in the *DAI\_UserCommandLoop*:
  - **CHANNEL\_ID\_HANDLE** - the channel ID of the channel, maps to the RTSP URI.

For the *DAI\_UserCommandAck\_Cnf* function, which has the following parameters:

- **TEMPORARY\_ID\_HANDLE** - the handle that was passed to the corresponding *DAI\_User\_CommandAck\_Req* function. If the implementation relies on RTSP for mapping this parameter, it can be generated from the (RTSP Session ID, Cseq RTSP field of the response) tuple.
- **RESPONSE\_CODE** - a response mapped from the RTSP response. The response mapping is described in Section 3.4.5.
- User-to-user data - this data can come as an attachment to the RTSP response, or may be created from the Range and Speed RTSP fields in the response, that may contain the actual values from the server.
- For each channel in the *DAI\_UserCommandLoop*:
  - **CHANNEL\_ID\_HANDLE** - the channel ID of the channel, mapped from the RTSP URI.

#### 3.4.2.4 Closing the channels: ChannelDelete

Closing the channels should be done with the **TEARDOWN** command. The streams can be **TEARDOWN**-ed one at a time, or if the **ChannelDelete\_Req** command contains all the channels in an RTSP session, the **TEARDOWN** can be applied to the control URI for that RTSP session. A complete command mapping (both client and server) is presented in Fig. 3.6.

Parameters mapping: The usage of DMIF parameters when using RTSP should be as follows:

For the *DAI\_ChannelDelete\_Req* function, which has the following parameters:

- For each channel in the *DAI\_ChannelHandleLoop*:
  - **REASON\_CODE** - a reason specific to DMIF. Can be ignored at RTSP level.
  - **RESPONSE\_CODE** - the response is not used in this function

For the *DAI\_ChannelDelete\_Cnf* function, which has the following parameters:

- For each channel in the *DAI\_ChannelHandleLoop*:
  - **REASON\_CODE** - the reason is not used in this function
  - **RESPONSE\_CODE** - a response mapped from the RTSP response. The response mapping is described in Section 3.4.5.

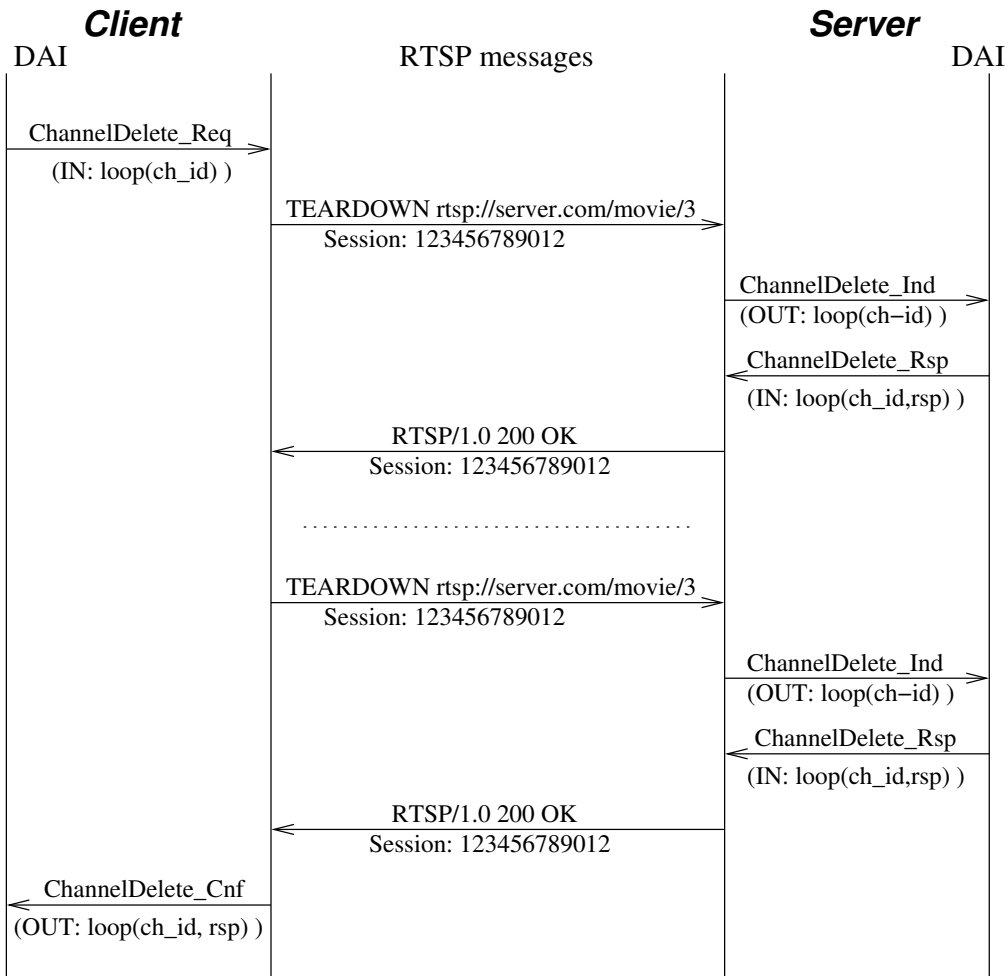


Figure 3.6: Mapping of ChannelDelete.

### 3.4.2.5 Service termination: ServiceDetach

A **ServiceDetach** should send a **TEARDOWN** to the control URI of all the remaining RTSP sessions, thus closing them all. A complete command mapping (both client and server) is presented in Fig. 3.7.

Parameters mapping: The usage of DMIF parameters when using RTSP should be as follows:

For the *DAI\_ServiceDetach\_Req* function, which has the following parameters:

- **SESSION\_ID\_HANDLE** - the DMIF Session ID, returned by the *DAI\_ServiceAttach\_Cnf* function
- **REASON\_CODE** - a reason specific to DMIF. Can be ignored at RTSP level

For the *DAI\_ServiceDetach\_Cnf* function, which has the following parameters:

- **SESSION\_ID\_HANDLE** - the DMIF Session ID of the closed session

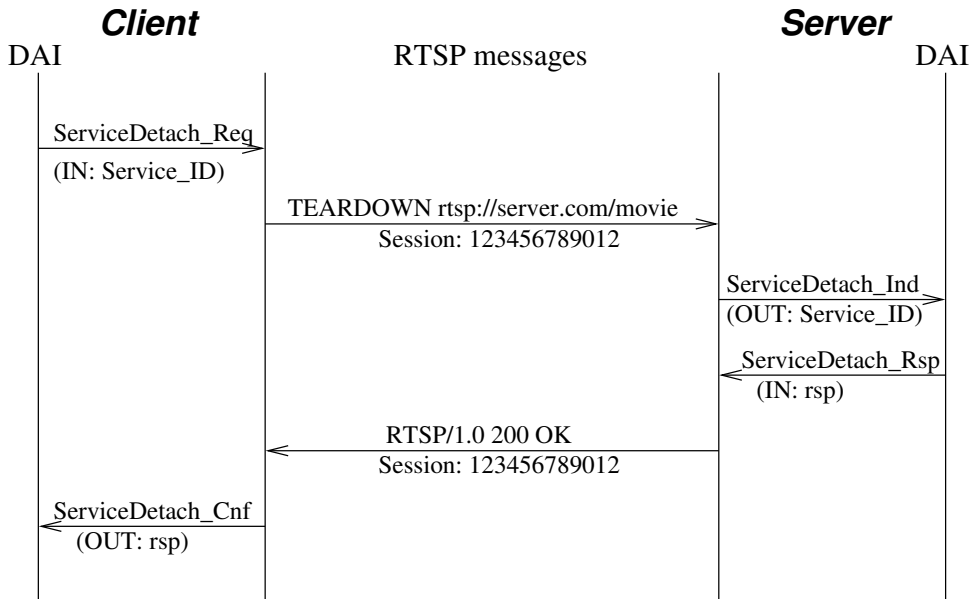


Figure 3.7: Mapping of ServiceDetach.

- `RESPONSE_CODE` - a response mapped from one of the RTSP responses. The response mapping is described in Section 3.4.5.

### 3.4.3 Mapping of RTSP commands to DMIF, on the server side

This section describes how RTSP commands received by a server can be translated to DMIF requests. This mapping assumes that the client is an RTSP/RTP client capable of requesting RTSP commands and receiving RTP packets. The server has a DMIF filter and loads a DMIF plugin to communicate with the RTSP/RTP client. The mapping describes how this plugin should handle RTSP requests and how to translate them into DMIF commands.

The reason why a server would like to use DMIF could be that it makes the transport/streaming part totally transparent for the application. Thus, a server can have several ways of data acquisition (e.g. from several file formats, from a live web camera, from a satellite TV broadcast) and several ways of streaming it (RTSP/RTP unicast/multicast, DMIF default signalling). DMIF and DAI can be the common denominator for all acquisition modes and transport modes, so that the server does not need to implement each transport mode for each acquisition mode.

#### 3.4.3.1 Service initialization: ServiceAttach

If the RTSP implementation is using a permanent connection (TCP) between the client and the server, a **DESCRIBE** command received at the server should trigger a `DAI_ServiceAttach_Ind` DMIF command. We assume here that the RTSP server is able to track subsequent commands coming from the same source, even if the RTSP session is not yet established.

Subsequent **DESCRIBE** commands coming from the same source/client should be ignored at the DMIF level.

A **SETUP** RTSP command coming before a **DESCRIBE** command should also trigger a DMIF *DAI\_ServiceAttach\_Ind* command. Normally, a **SETUP** command would trigger a *DAI\_ChannelAdd\_Ind* command, but at the DMIF level we need to create a session first.

The DMIF session should be closed if the connection with the client is lost.

In an RTSP implementation using a non-permanent connection, we cannot track commands coming from the same source after we received a **DESCRIBE** request because only the first **SETUP** request will create an RTSP session. In this case we do nothing at the DMIF level when receiving a **DESCRIBE** request, but the RTSP level should be capable of sending all the initialization information to the client. In this case, a **SETUP** request that creates an RTSP session will also trigger a *DAI\_ServiceAttach\_Ind*.

A complete command mapping (both client and server) is presented in Fig. 3.3.

Parameters mapping: The usage of DMIF parameters when using RTSP should be as follows:

For the *DAI\_ServiceAttach\_Ind* function, which has the following parameters:

- **SESSION\_ID\_HANDLER** - a Session ID generated by DMIF (can be a pointer to the session instance in a C++ implementation or a pointer to the session data structure in a C implementation). There should be a one-to-one correspondence between the DMIF Session ID and the RTSP Session ID.
- **Service\_name** - the name of the service (DMIF plugin) used in this session. This name is implementation dependent, may give a hint that RTSP/RTP are used for DMIF signalling (e.g. "RTSP/RTP DMIF Plugin").
- **User-to-user data** - if the **DESCRIBE** request contained attachments, they are transmitted to the DMIF server as user-to-user data.

For the *DAI\_ServiceAttach\_Rsp* function, which has the following parameters:

- **RESPONSE\_CODE** - a response to be send to the client. It will be mapped to the RTSP response. The response mapping is described in Section 3.4.5.
- **SESSION\_ID\_HANDLE** - the Session ID that was received by the peer *DAI\_ServiceAttach\_Ind* function
- **User-to-user data** - should contain the IOD. The IOD should be sent to the client, using one of the three methods described in Section 3.4.2.1. If this **ServiceAttach** comes as a result of a **SETUP** command without a **DESCRIBE**, the IOD should be ignored by the RTSP layer.

### 3.4.3.2 Opening channels: ChannelAdd

A **SETUP** received at the server should trigger a **ChannelAdd\_Ind** command. If there was no **ServiceAttach** DMIF command, the **SETUP** command should also trigger it (see Section 3.4.3.1 for more details). A complete command mapping (both client and server) is presented in Fig. 3.4.

Parameters mapping: The usage of DMIF parameters when using RTSP should be as follows:

For the *DAI\_ChannelAdd\_Ind* function, which has the following parameters:

- **SESSION\_ID\_HANDLE** - the session identifier, the DMIF correspondent of the RTSP session
- The *DAI\_ChannelAddIndLoop* contains only one channel (we do not wait for more than one **SETUP** requests to come to the server) with the following parameters:
  - **CHANNEL\_ID\_HANDLE** - the channel ID, generated by DMIF. DMIF should also keep a one-to-one mapping between the RTSP URI of a stream, the channel ID and the **ES\_ID** for a stream.
  - *DAI\_ChannelDescriptor* - this descriptor contains QoS parameters. These parameters cannot be mapped from any RTSP command, field or attribute. Hints about their usage are given in Section 3.4.4.
  - **CHANNEL\_DIRECTION** - DMIF specifies three directions: upstream, downstream, bi-directional and unspecified. The direction can be mapped from the "mode" parameter of the Transport field of the **SETUP** RTSP command. A "mode=play" will map to a downstream direction, a "mode=record" will map to an upstream direction and a "mode=play, record" will map to a bi-directional stream. If there is no mode parameter in the Transport field then the direction will be the DMIF unspecified direction.
  - User-to-user data - this data should contain the **ES\_ID** of the stream that is requested.

For the *DAI\_ChannelAdd\_Rsp* function, which has the following parameters:

- **SESSION\_ID\_HANDLE** - the session identifier, the DMIF correspondent of the RTSP session, mapped to the RTSP Session ID
- The *DAI\_ChannelAddRspLoop* contains only one channel with the following parameters:
  - **RESPONSE\_CODE** - a response to be send to the client. It will be mapped to the RTSP response. The response mapping is described in Section 3.4.5.

- CHANNEL\_ID\_HANDLE - the channel ID, not used, because the RTSP does not need a URI in the response.
- User-to-user data - other data that can be send to the client as an attachment to the **SETUP** response.

### 3.4.3.3 Controlling the channels: UserCommand

A **PLAY** or **PAUSE** command received at the server should trigger a **UserCommandAck\_Ind** command. If the URI of the RTSP command identifies a media stream, the DMIF loop will contain only the corresponding channel. If the RTSP command was issued to the control URI of the RTSP session, the DMIF command will contain all the channels in the RTSP/DMIF session. A complete command mapping (both client and server) is presented in Fig. 3.5.

Parameters mapping: The usage of DMIF parameters when using RTSP should be as follows:

For the *DAI\_UserCommandAck\_Ind* function, which has the following parameters:

- TEMPORARY\_ID\_HANDLE - temporary handle for the command, used internally by DMIF. However, if the server supports concurrent DMIF/RTSP commands, this parameter can be mapped from the (RTSP Session ID, Cseq RTSP field of the request) tuple.
- User-to-user data - this data contains the actual command and its parameters. The data format is custom and should be generated from the received RTSP request. The user data may contain:
  - One of the two supported commands, **PLAY** or **PAUSE**, mapped directly from the RTSP command request.
  - A playing position or range, or pausing position. The position or range can be generated from the Range RTSP field, if any.
  - A playing speed (for **PLAY** commands only). This can be generated from the Speed RTSP field.
- For each channel in the *DAI\_UserCommandLoop*:
  - CHANNEL\_ID\_HANDLE - the channel ID of the channel, mapped from the RTSP URI.

For the *DAI\_UserCommandAck\_Rsp* function, which has the following parameters:

- TEMPORARY\_ID\_HANDLE - temporary handle for the command, used internally by DMIF, the same that was passed to the corresponding *DAI\_UserCommandAck\_Ind* function. If the server supports concurrent DMIF/RTSP commands, this parameter can be mapped to the (RTSP Session ID, Cseq number of the RTSP response) tuple.

- **RESPONSE\_CODE** - a response to be send to the client. It will be mapped to the RTSP response. The response mapping is described in Section 3.4.5.
- User-to-User data - this data might contain the values of the actual playing or stopping position and the actual speed value that was used by the server. If this is the case, the position should be mapped to the Range RTSP field of the response, and the speed should be mapped to the Speed RTSP field of the response.
- For each channel in the *DAI\_UserCommandLoop*:
  - **CHANNEL\_ID\_HANDLE** - the channel ID, not used, because the RTSP does not need a URI in the response.

### 3.4.3.4 Closing the channels: **ChannelDelete**

A **TEARDOWN** for a media stream URI will trigger a **ChannelDelete\_Ind** command for that stream. The command will contain only the channel corresponding to that stream. A complete command mapping (both client and server) is presented in Fig. 3.6.

Parameters mapping: The usage of DMIF parameters when using RTSP should be as follows:

For the *DAI\_ChannelDelete\_Ind* function, which has the following parameters:

- For the channel in the *DAI\_ChannelHandleLoop*:
  - **REASON\_CODE** - a reason specific to DMIF. Because there is no RTSP equivalent for this parameter, it should have the value **REASON\_OK**.
  - **RESPONSE\_CODE** - the response is not used in this function

For the *DAI\_ChannelDelete\_Rsp* function, which has the following parameters:

- For the channel in the *DAI\_ChannelHandleLoop*:
  - **REASON\_CODE** - the reason is not used at RTSP level.
  - **RESPONSE\_CODE** - a response to be send to the client. It will be mapped to the RTSP response. The response mapping is described in Section 3.4.5.

### 3.4.3.5 Service termination: **ServiceDetach**

A **TEARDOWN** for the control URI of the RTSP session should trigger a **ServiceDetach\_Ind**. The loss of the RTSP session (e.g. timeout) should also trigger a **ServiceDetach\_Ind**. A complete command mapping (both client and server) is presented in Fig. 3.7.

Parameters mapping: The usage of DMIF parameters when using RTSP should be as follows:

For the *DAI\_ServiceDetach\_Ind* function, which has the following parameters:

- `SESSION_ID_HANDLE` - the session identifier, the DMIF correspondent of the RTSP session, mapped to the RTSP Session ID
- `REASON_CODE` - a reason specific to DMIF. Because there is no RTSP equivalent for this parameter, it should have the value `REASON_OK`.

For the *DAI\_ServiceDetach\_Rsp* function, which has the following parameters:

- `SESSION_ID_HANDLE` - the session identifier, the DMIF correspondent of the RTSP session, mapped to the RTSP Session ID.
- `RESPONSE_CODE` - a response to be send to the client. It will be mapped to the RTSP response. The response mapping is described in Section 3.4.5.

### 3.4.4 Usage of the `DAI_ChannelDescriptor` QoS parameters when using RTSP as control protocol

RTSP alone cannot handle QoS parameters. The client can send them to the server as attached data to the **SETUP** request. Use of these parameters may be possible when using QoS for IP networks (Differentiated Services, Integrated Services). If the server needs QoS parameters and they were not sent by the client, it should create them using some default values.

### 3.4.5 Mapping between DMIF response and RTSP response

There are no normative values for the DMIF response at the time of the design of this mapping. The following DMIF response values are taken from IM1.

DMIF response	RTSP response
<code>RESPONSE_OK</code>	200 OK
<code>RESPONSE_SERVICEDLL_NOTFOUND</code>	503 Service Unavailable
<code>RESPONSE_SERVICEDLL_ERROR</code>	501 Internal Server Error
<code>RESPONSE_URI_NOT_RECOGNIZED</code>	400 Bad Request
<code>RESPONSE_CHANNEL_NOT_FOUND</code>	404 Not Found
<code>RESPONSE_RESOURCES_EXHAUSTED</code>	501 Internal Server Error
<code>RESPONSE_REMOTE_ERROR</code>	501 Internal Server Error
<code>RESPONSE_URI_NOT_FOUND</code>	404 Not found
<code>RESPONSE_END_OF_DATA</code>	501 Internal Server Error
<code>RESPONSE_UNKNOWN_ERROR</code>	501 Internal Server Error

### 3.4.6 Interoperability and optimization issues for DMIF clients and servers which use RTSP for signalling

This section proposes modifications to the previous mappings that are necessary in order that a DMIF client and a DMIF server that both use RTSP for signalling understand each other better. The benefits of these modifications are:

1. One-to-one mapping between a DMIF session on the client side and a DMIF session on the server side. Using the previous mappings, a DMIF session on the client side can contain more than one RTSP sessions (one for each group of streams having the same time basis), but an RTSP session on the server side corresponds to one DMIF session. Thus, it is possible that one DMIF session on the client side corresponds to several sessions to the server side. The benefit of having this one-to-one correspondence is that the calling of several useless **ServiceAttach** and **ServiceDetach** commands on the server side is avoided. This benefit requires one additional RTSP extension field for each RTSP request and answer. Implementing these modifications also lets the client and the server know that their peer is DMIF aware and also knows these modifications.
2. The possibility to carry the DMIF REASON parameter. In order to have this benefit an additional RTSP extension field is required for each **TEARDOWN** RTSP request.
3. The possibility to send simultaneously a **PLAY** or **PAUSE** commands to several random streams (on the same server), independent of their time basis. This command requires a customized URI in the RTSP request. This URI would be most probably not understood by a server that is not aware of these modifications, so the answer will most probably be an RTSP error URI\_NOT\_FOUND.

These benefits and modifications can be implemented independent one of each other, however, implementing the modifications required by 3 without implementing 1 may lead to a situation when the client sends the customized URI described in 3 to a server that does not know these modifications. If the modification proposed in 1 is not implemented, then the server and the client will not know if the peer knows DMIF or not.

It is safe to send a request or an answer containing an RTSP extension field to a standard RTSP server or client. These fields will be ignored by the standard implementation. (RTSP RFC [118] - Section 1.5 Extending RTSP, page 11).

#### 3.4.6.1 Modifying the client and server in order to have one-to-one mapping of the DMIF server

In order to have one-to-one mapping of the DMIF session, a new RTSP extension field has to be introduced. We propose this field to be called: "DMIF-Session" and its format should be:

```
DMIF-Session = "DMIF-Session" ":" session-id
```

The session-id should be an alphanumeric string that would identify a DMIF Session ID. Example:

```
DMIF-Session: DMI1234565F
```

The Session ID should have the same value for all the messages that belong to the same DMIF session, even if they are carried in different RTSP sessions.

The DMIF-Session field should be carried in all the RTSP requests and answers of the DMIF session.

The added benefit relies on the fact that the first **DESCRIBE** request that comes from the client will contain the DMIF-Session field. If the server is not DMIF aware, the field will be ignored and the server will not send a DMIF-Session field in his answer. In this case, the client will know that the server is not DMIF aware. If a DMIF aware server receives a request without a DMIF-Session field, it knows that the client is not DMIF aware. If the client sends a DMIF-Session in its **DESCRIBE** request and the DMIF-Session field also appears in the **DESCRIBE** response, then both the client and the server will know that the other one is DMIF aware. In this case they can use the other two extensions.

On the client side, the implementation of this benefit requires only adding this extension field to the client requests, and checking the existence of the field in the responses coming from the server.

On the server side, the implementation of this benefit requires adding the extension in responses if the extension field was found in the request. Beside this, few modifications have to be added to the mapping of RTSP commands to DMIF commands:

1. Receiving a **DESCRIBE** or a **SETUP** RTSP command with a new session-id in the DMIF-Session field would trigger a **ServiceAttach\_Ind** command.
2. No subsequent command having the same DMIF-Session should trigger a **ServiceAttach\_Ind**, even if they create a new RTSP session.

The consequence of these two modifications is that a second **DESCRIBE** requests having the same DMIF-Session will be ignored at the DMIF level. A **SETUP** command can trigger a **ServiceAttach\_Ind**, this is the case when the client knows already all the initialization information and does not need to send a **DESCRIBE** request.

### 3.4.6.2 Modifying the client and the server in order to carry a DMIF REASON

DMIF states that a reason should be given when a channel is closed or the session is closed (service detach). This reason parameter does not have an RTSP correspondence, so a new RTSP field has to be created for carrying it. We propose this parameter to be called: "DMIF-Reason" and its format should be:

```
DMIF-Reason = "DMIF-Reason" ":" reason
```

The reason is an alphanumeric string and has one of the values: OK, UNEXPECTED\_ERROR, etc. These values should map one-to-one to the values actually used by DMIF.

Example:

```
DMIF-Reason: OK
```

Only the **ChannelDelete** and **ServiceDetach** DMIF commands carry a reason. They both map to the **TEARDOWN** RTSP commands, so only this command should carry the DMIF-Reason field.

On the client side, the implementation of this benefit requires only adding this extension field to the client requests for the **TEARDOWN** RTSP commands.

On the server side, the implementation of this benefit requires the parsing of the extension from the **TEARDOWN** requests. In this case, the `REASON_CODE` passed to the `DAI_ChannelDelete_Ind` and `DAI_ServiceDetach_Ind` functions will not be set to `REASON_OK` by default, but will be the result of parsing the DMIF-Reason field.

### 3.4.6.3 Modifying the client and the server in order to allow one **PLAY** and **PAUSE** user commands to be send to an arbitrary set of streams

In order to identify more than one stream in a **PLAY** or **PAUSE** RTSP stream we need to specify them somehow in the URI. The proposed method is to specify the channel IDs as data in the URI. Thus, the URI should be composed by the control URI of one of the RTSP sessions in the DMIF session followed by the channel IDs numbers, as data (similar with HTTP GET method).

Example: In order to specify the channels with the IDs 5, 3 and 7 we construct the following URI (assuming that `rtsp://server.com/movie` is the control URI of one of the composing RTSP sessions):

```
rtsp://server.com/movie?ch_id=5&ch_id=3&ch_id=7
```

In this way the DMIF layer can identify the channels for which the command was issued.

If this command is send to a server that does not understand it, the server will take no action, and an error code will be returned.

The added benefit is that any number of streams, belonging to an RTSP sessions can be issued a command synchronously and without added delays due to round-trip delays.

On the client side, the implementation of this benefit requires the implementation of the special URI construction.

On the server side, the implementation of this benefit requires parsing the URI and forcing the RTSP part not to send an error response due to it.

Additional care has to be taken if the command fails for one or more of the channels. We can only send one response back using the RTSP mechanism. If the command fails for one or more channels, the DMIF layer should force all of the channels to fail, because there is no way to tell to the client which of the individual channels failed. (We can add an additional field to the response that would specify the failing channels).

This method allows the creation of only one RTSP session for each DMIF session (instead of many RTSP sessions, one for each group of streams having the same time basis). The reason for this is that using this method, any command such as **PLAY**

or **PAUSE** can be issued for an arbitrary set of channels, thus there is no need for several separate RTSP sessions.

### 3.4.7 A previous version of mapping of DMIF primitives to RTSP methods

A previous mapping of DMIF to RTSP was presented in various aspects and at different stages of its development in [80, 20, 12, 21]. This section describes the differences between that version and the mapping presented in this chapter.

The DMIF primitive that was modified in the mapping process is **ServiceAttach**. The previous version maps **ServiceAttach** to two RTSP methods: a **DESCRIBE** followed by a **SETUP**. The complete command mapping is shown in Fig. 3.8.

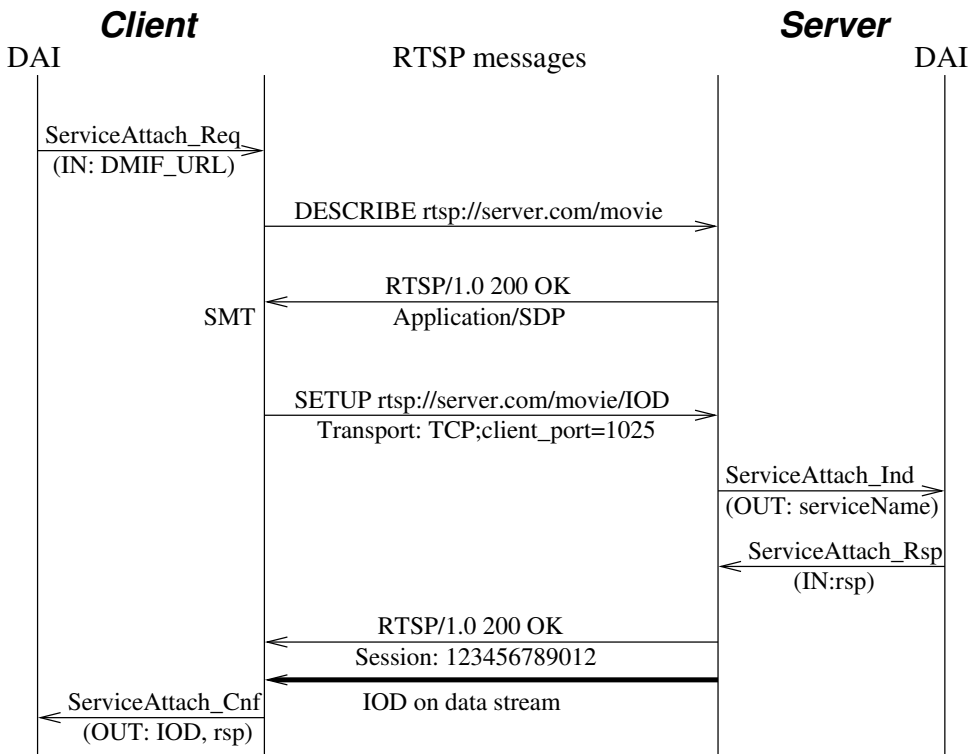


Figure 3.8: Previous version of ServiceAttach mapping to RTSP.

The **DESCRIBE** RTSP command is used to retrieve transport information, in SDP format, about all the Elementary Streams involved in the DMIF session. For each Elementary Stream the SDP specified an RTSP control URI and the ES\_ID of that Elementary Stream, as a media attribute. The control URI for each channel is generated from the RTSP session control URI followed by the DMIF channel ID. Thus, for each Elementary Stream we have a complete, one to one, mapping between the ES\_ID (identifying the stream at Sync Layer level), Channel ID (identifying the stream at DMIF level) and the RTSP URI (identifying the stream at RTSP level):

```
m=audio 0 RTP/AVP 99
```

```
a=rtpmap:99 mpeg4-sl/1000
a=control:rtsp://localhost/linda.mp4/2
a=control:rtsp://localhost/linda.mp4/2
```

The control attribute (third line) specifies the RTSP URI and at the same time, the DMIF channel: 2. The x-mp4-esid attribute (fourth line) specifies the ES\_ID: 2113.

The second RTSP method (**SETUP**) is used to retrieve the IOD. The client issues a **SETUP** request for a special RTSP URI, which the server recognizes as the IOD URI. This URI is obtained by adding /IOD to the RTSP session control:

```
rtsp://localhost/linda.mp4/IOD
```

This IOD stream is created using TCP as transport protocol, for greater reliability. The IOD channel, even if only used in the beginning of the session, stays open until the end of the DMIF session. This prevents the RTSP session to close in case the DMIF layer closes all active channels and opens others.

There are two disadvantages associated with mapping the **ServiceAttach** to two RTSP methods:

- Setting up the DMIF session takes two RTD (round trip delays).
- The TCP connection used to establish the channel for the IOD transfer will just waste system resources after the IOD is transferred until the end of the session. While this waste of resources is negligible on the client side, it may become significant on the server side, for a high number of connections.

The optimizations and modifications presented in Section 3.4.6 can also be applied to this older mapping.

## 3.5 Implementation

The mapping described in Section 3.4 was implemented in order to prove its concept. Two systems, based on different servers were developed:

1. The first system uses a server that is DMIF-aware and uses RTSP signalling and RTP for streaming the contents of mp4 files [39]. The concept of this system is illustrated in Fig. 3.9.
2. The second system uses an RTSP server that is not DMIF-aware but is able to stream mp4 files (mp4 [39] is the file format adopted by the MPEG-4 standard, it is similar with QuickTime). The concept of this system is illustrated in Fig. 3.10.

As client, both systems use IM1-2D [41], developed by the IM1 group inside the MPEG-4 community. This group was activated in April 1997 to verify the functionality of the MPEG-4 Systems specification. The IM1 software is freeware and will be included in the MPEG-4 reference software. IM1 includes a DMIF Filter and an

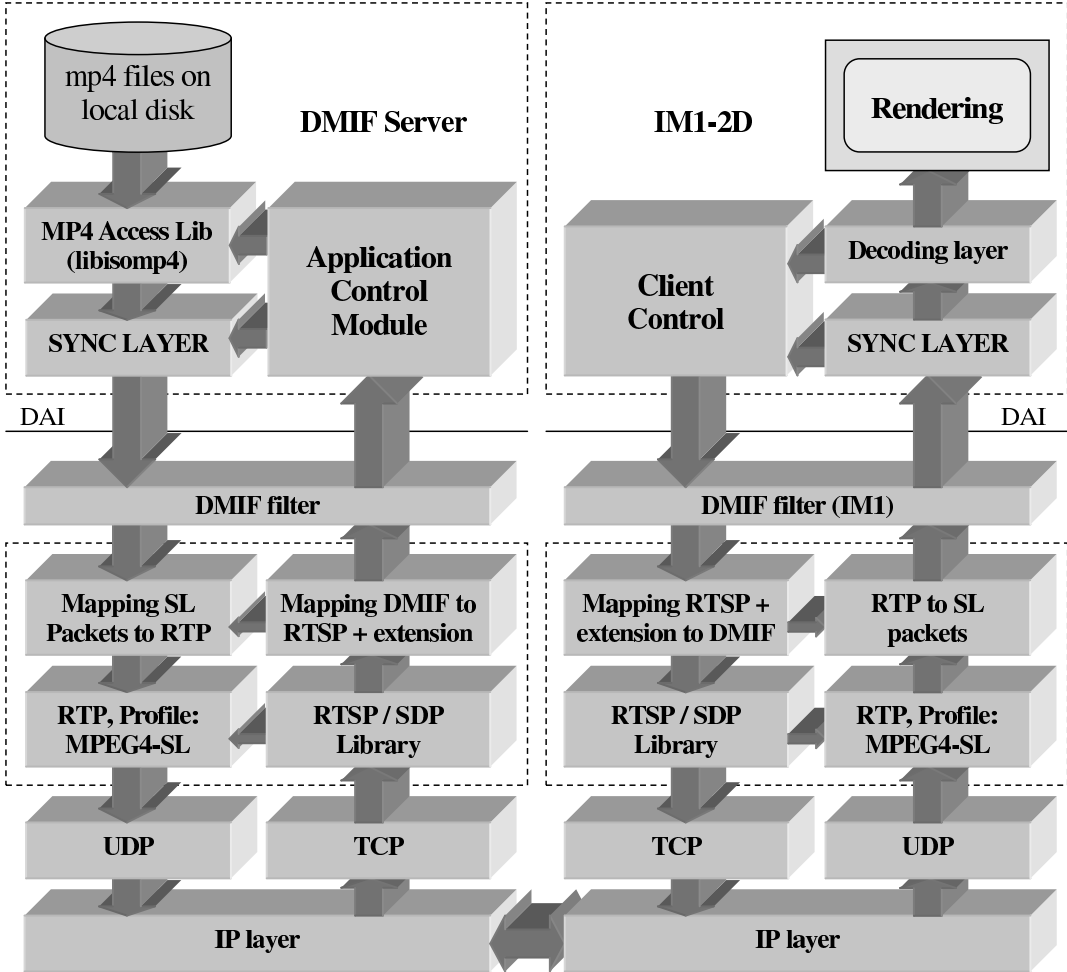


Figure 3.9: MPEG-4 System based on a server with RTSP-based DMIF instance.

MPEG-4 FlexDemux, as well as a DMIF instance for accessing streaming content from local files, in both a so-called TRivial File format (TRIF), defined internally to IM1, and MPEG-4 File Format [39]. The chosen version of IM1 was 3.8, which was the latest version available at the moment of starting the implementation work.

The only modification of IM1-2D was adding a DMIF plugin based on RTSP/RTP/SDP and the mapping described in Section 3.4. We have implemented this filter on top of the RTSP, RTP, SDP library described in Chapter 2. The RTP profile used for transport of MPEG-4 elementary streams is described in [10, 9, 11, 12].

The mapping implementation is able to issue one RTSP command for one stream only. The DMIF channel IDs are the same as ES\_IDs and are specified as the last part of the URI's path. The SDP attachment is generated by DMIF layer on the fly, using additional information: the handle to the opened mp4 file. Thus, DMIF is able to query the mp4 file for all its streams and generate a complete SDP information structure. The **ServiceAttach** is mapped to one RTSP **DESCRIBE** and the IOD

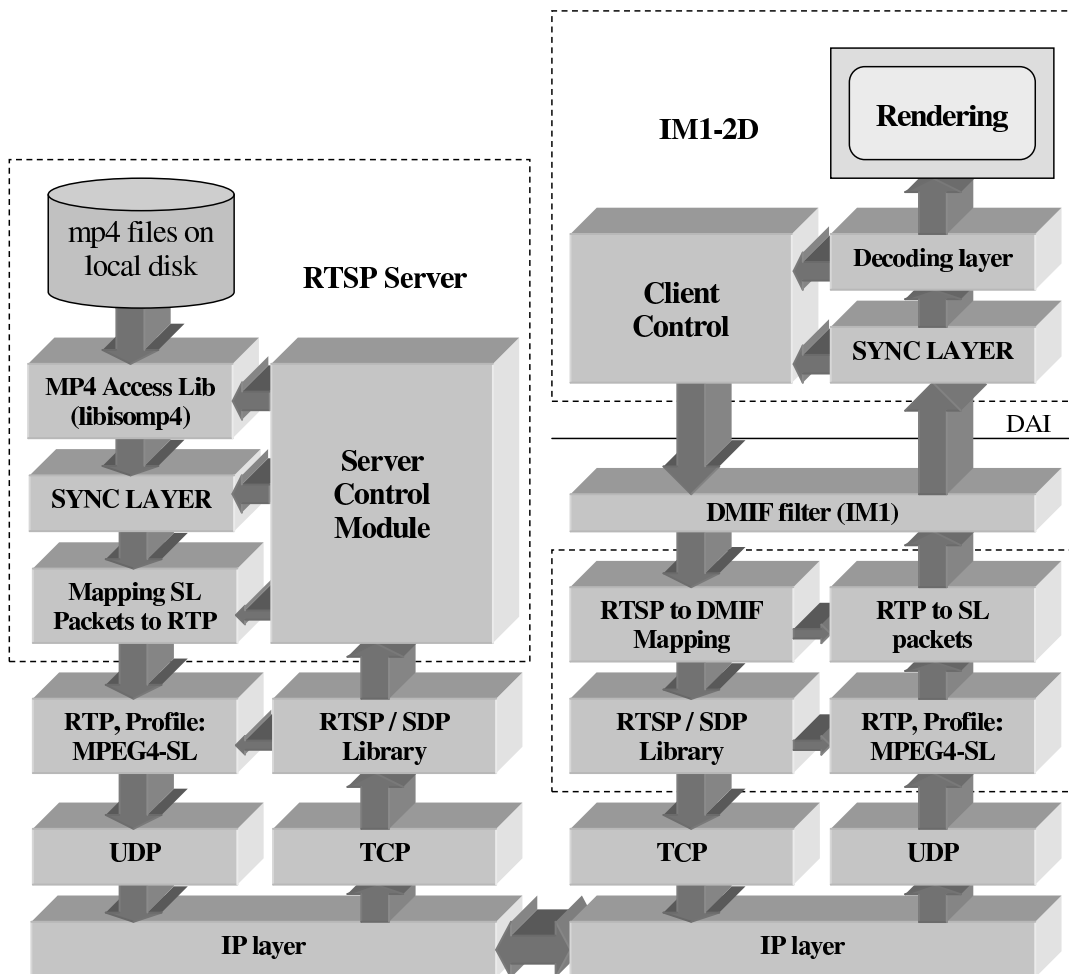


Figure 3.10: MPEG-4 System based on a server with RTSP-only signalling.

is sent to the client embedded in the SDP attachment. The DMIF-based server also implements the DMIF-Session and DMIF-Reason additional fields.

### 3.5.1 Performance evaluation

In order to measure the performance of our implementation, we created a log file where we print the time when certain events do happen. The log file is operated by the DMIF filter of the client (IM1). The time counting starts when a DMIF plugin is selected and the time accuracy is in milliseconds (the measurements were performed under Windows 2000 and the accuracy of the operating system seems to be 10 ms). The PC used for the measurements was a Pentium 4 running at 1.5 GHz with 256 MB of RAM.

Using this log file we intend to measure how much extra time is required for the RTSP DMIF plugin as compared with a local file plugin. For this, we play the same MPEG-4 presentation (Linda, a well-known MPEG-4 sample media, containing

audio and video streams) from both local file (mp4), using the MP4 DMIF plugin, and then using the RTSP DMIF plugin. The server resides on the same machine. This makes the measurements more accurate, since some operations used in both cases are performed on the same machine (like the opening of the mp4 file).

The order of operations (function calls) and their timing is presented in the table below:

Step	Event	File (s)	RTSP (s)	Diff.
1	DMIF plugin selected	0	0	0
2	<i>DAI_ServiceAttach_Req</i> called	0	0	0
3	<i>DAI_ChannelAdd_Req</i> called	0.081	0.150	0.069
4	<i>DAI_ChannelAdd_Req</i> ended	0.081	0.160	0.079
5	<i>DAI_ServiceAttach_Req</i> ended	0.081	0.160	0.079
6	<i>DAI_UserCommand_Req</i> called	0.091	0.160	0.069
7	<i>DAI_UserCommand_Req</i> ended	0.091	0.160	0.069
8	<i>DAI_UserCommand_Req</i> called	0.091	0.160	0.069
9	<i>DAI_UserCommand_Req</i> ended	0.091	0.160	0.069
10	<i>DAI_ChannelAdd_Req</i> called	0.111	0.190	0.079
11	<i>DAI_ChannelAdd_Req</i> ended	0.111	0.200	0.089
12	<i>DAI_UserCommand_Req</i> called	0.111	0.200	0.089
13	<i>DAI_UserCommand_Req</i> ended	0.111	0.200	0.089
14	<i>DAI_UserCommand_Req</i> called	0.121	0.200	0.079
15	<i>DAI_UserCommand_Req</i> ended	0.121	0.200	0.079
16	<i>DAI_ChannelDelete_Req</i> called	294.573	294.573	0
17	<i>DAI_ChannelDelete_Req</i> ended	294.573	294.573	0
18	<i>DAI_ChannelDelete_Req</i> called	294.573	294.573	0
19	<i>DAI_ChannelDelete_Req</i> ended	294.573	294.573	0
20	<i>DAI_ChannelDelete_Req</i> called	294.573	294.573	0
21	<i>DAI_ChannelDelete_Req</i> ended	294.573	294.583	0.010
22	<i>DAI_ChannelDelete_Req</i> called	294.573	294.583	0.010
23	<i>DAI_ChannelDelete_Req</i> ended	294.573	294.583	0.010
24	<i>DAI_ServiceDetach_Req</i> called	294.573	294.583	0.010
25	<i>DAI_ServiceDetach_Req</i> ended	294.573	294.593	0.020

The selection of the DMIF plugin by the DMIF filter is considered to be the time origin of the measurements. The first function called by the client is *DAI\_ServiceAttach\_Req*. The DMIF implementation in IM1 is done in such a way that the *\_Req* function calls do not return until the full action has take place and the corresponding *\_Cnf* function has been called and returned. Inside the *DAI\_ServiceAttach\_Cnf* function the *DAI\_ChannelAdd\_Req* function is called. This way we can explain the call and return of the *DAI\_ChannelAdd\_Req* at steps 3 and 4 in the table above, before the *DAI\_ServiceAttach\_Req* function returns. The complete function calling scenario is as follows:

- The *DAI\_ServiceAttach\_Req* function is called to retrieve the IOD for the

“Linda” presentation (step 2).

- The IOD is returned to the *DAI\_ServiceAttach\_Cnf* function (somewhere between step 2 and step 3). Inside this function the IOD is parsed and it is revealed that it refers to two other streams: the OD and BIFS streams of the “Linda” presentation. The function *DAI\_ChannelAdd\_Req* is called to retrieve these streams (step 3). Then the *DAI\_ChannelAdd\_Cnf* function is called, it returns, and we are back inside the *DAI\_ChannelAdd\_Req* function. We return from this function back to *DAI\_ServiceAttach\_Req* function (step 4), which itself returns (step 5).
- The client application is issuing “**PLAY**” commands for the BIFS and OD streams, for each separately, at steps 6, 7, 8 and 9.
- The OD stream is received and parsed, thus the application finds out about two other streams: a video stream and its corresponding audio stream. At step 10 the application issues a *DAI\_ChannelAdd\_Req* to retrieve both these streams. At step 11 the *DAI\_ChannelAdd\_Req* function returns.
- The client application is issuing “**PLAY**” commands for the video and audio streams, for each separately, at steps 12, 13, 14 and 15. Now we have the player (IM1) playing our presentation (between step 15 and step 16).
- After about 294 seconds the “Linda” presentation is ending and the application asks DMIF to close all channels and close the session.
- Each of the 4 channels is closed separately at steps 16 to 23.
- At steps 24 and 25 the session is closed.

The main difference between the MP4 DMIF plugin and the RTSP DMIF plugin and its associated server is that all the action that is happening inside the MP4 DMIF plugin (opening the requested mp4 file, retrieving information about its streams, reading the streams) is happening inside the server for the RTSP DMIF plugin. Thus the time difference between similar events in the two plugins is time taken by the DMIF to RTSP mapping, RTSP signalling, network connection establishment and streaming and switching between processes (in order to complete a command, e.g. **ServiceAttach**, switching to the server process and back to the client process is required). While we are only interested in the time taken by the DMIF to RTSP mapping and RTSP signalling, measuring this time only, without the operating system dependent part (networking and process switching), is very difficult to obtain.

We can see from the table above that a big time difference (about 70 ms) appears in the beginning while processing the **ServiceAttach** request, before we reach the first **ChannelAdd** request (step 3). Because there is only one RTSP method called up to this point (**DESCRIBE**), this difference can be explained by the additional operations required to create the SDP attachment on the server side: the querying of the mp4 file for all its streams and the creation of the SDP itself (neither operation takes place for the MP4 DMIF plugin). From step 3 until step 15, when all the

preparation for playing ends, we encounter some more delay, in the order of 10 ms. The main reason for this delay are the 8 RTSP commands exchanged between the client and the server.

The session closing (steps 16 to 25) takes no measurable amount of time for the MP4 DMIF plugin, however is in the range of 10 to 20 ms for the RTSP DMIF plugin, for which closing the session requires 5 RTSP commands.

The performance data show that while the RTSP/DMIF based server is able to accept around 5 connections per second, it is able to process hundreds of RTSP commands per second. The number of accepted connections per second can be increased if the SDP creation is optimized (including creating it off-line).

The complete log for RTSP messages exchanged between the client and the server and their timings is presented in Annex A.

### **3.6 Summary and author's contributions**

In this chapter we addressed issues related with the interoperability between existing, deployed multimedia applications, using IETF protocols for remote retrieval (RTSP, SDP and RTP) and emerging MPEG-4 applications, using DMIF for remote retrieval. The proposed solution replaces DMIF Default Signalling with RTSP, SDP and RTP by mapping DMIF primitives to these protocols.

The scientific contribution of this approach is the mapping between DMIF and the IETF protocols. The implementation of this approach was successfully demonstrated at the 50th MPEG meeting in Maui, Hawaii, in December 1999. At that time, this architecture was one of the first two examples of a complete system for the transport of MPEG-4 content over the Internet based on mapping onto IETF protocols.

The work presented in this chapter is based on [20, 80, 12]. The contribution of the author to this work concerns the control part. The transport part mapping was done by authors of RFCs and Internet Drafts describing profiles for carrying MPEG-4 over RTP [45, 46, 133, 66, 10, 130]. Several people helped with the implementation: Serkan Kiranyaz (the transport part in the first versions), Valentin Gheorgita (support for mp4 file format), Marius Vlad (original mapping of RTSP to DMIF, described in Section 3.4.7). Even if some of the problems and the detailed mapping of the control part are original work, many issues and solutions were raised during MPEG or IETF standardization meetings or other brainstorming meetings. Acknowledged for their contributions are Roberto Castagno, Marius Vlad, Irek Defée, Jami Kangasoja, Mika Rustari, Andrea Basso, Dave Singer, Guido Franceschini and Socrates Varakliotis.

## Chapter 4

# The Architecture of a Multimedia Player

In this chapter we describe the design and implementation of a multimedia client for playing high-quality multimedia content. The multimedia player is primarily intended to allow RTP/RTSP streaming of high-quality content over broadband links, and playback it on a Linux-based Set-Top Box (STB). The main goals of this multimedia player are:

- to allow the playback of streams from the network on a Linux PC or a Linux-based STB. When playing on a Linux PC, the multimedia player should use software decoding. When playing on a Linux-based STB, the player may also use the MPEG-2 hardware decoder that is available, through its API, on the STB.
- to be able to support several types of inputs (file, VoD, multicast) and several types of compression formats (MPEG-1, MPEG-2, MPEG-4, etc).
- to provide an effective user interface: graphical user interface on Linux PC and remote control interface on a Linux STB.

Because of these goals, our multimedia player is designed in such a way that all transport, demultiplexing, decoding, input and output functionalities are implemented as external plugins (libraries with clear interface, which are loaded at runtime if they are needed). By having the input commands implemented as plugins, the player can provide several interfaces to the user. One of these interfaces may be a remote-control: if the remote control hardware is found, the player loads the input plugin handling the remote control. By having the media processing implemented as plugins, the player can have several possible ways of handling the same media. For example, if there is hardware support for decoding a certain compression format, the player can use the plugin that interfaces that hardware. If there is no such hardware available, the player will use a plugin that performs the decoding in software.

## 4.1 Motivation

At the beginning of the year 2000 it became clear that delivering video and audio to the customers, using broadband network (IP over xDSL or Cable Modems) was not only technically possible, but could become cost-effective too. But to achieve this cost-effectiveness, several technologies involved in the content delivery chain had to be cost-effective. The receiving and playback equipment required on the client side, in people's homes, was among them. For large scale implementations of such systems it was considered that the highest burden for the final price the consumers have to pay was carried by this playback equipment at the client side. This equipment was expensive compared with other consumer electronics devices. There are two ways to increase the value/cost ratio:

1. Decrease the cost of the playback equipment to be comparable to other consumer electronic devices.
2. Increase the perceived value of the equipment for the client by adding additional features (games, music, DVD, email, web browsing, etc.).

One way to put all the above mentioned applications into a single set-top box is to have it based on a PC architecture and software, since a PC is able to run all these applications. For the set-top box to be effective, both its hardware and its software have to be effective, hence Linux operating system and applications licensed using the GNU Public License have a potential advantage over other operating systems that require a non-free license for each system and application instance.

Few years ago, many applications were available for the Linux operating system: email clients, web browser, games, music player. However, a good video player, capable of playing MPEG-2 Transport Stream (TS) send using RTP [40, 54, 52] and controlled using RTSP was missing as a Linux application. MPEG-2 TS is the format in which Digital TV streams are sent. Significant amount of content exists in this format, all the necessary infrastructure is in place, and the cost of STB supporting MPEG-2 was significantly cheaper than the cost of STB supporting MPEG-4.

Because existing Linux video player projects were not promising enough at that time, it was decided to start from scratch and design such a player.

## 4.2 History and analysis of Linux video players

In early 2000 (top of the line PC was 500-600 MHz Pentium III), Windows was a more suitable operating system than Linux for video playback. Most graphic cards had drivers supporting Microsoft DirectX [91], able to do hardware scaling of the displayed video, color space conversion (YUV to RGB) and in some cases hardware-based IDCT and motion compensation. MPEG-1 and VCD playback in Windows was common, DVD playback (MPEG-2) was possible with the help of additional decoding cards. DivX [96] was appearing and its popularity was increasing rapidly. At the same time, in Linux, DVD playback was almost impossible and popular codecs

remained unsupported, mostly because of the absence of a clear video playback system like DirectShow [92, 73], or because of patents. The XFree86 version 3.x server was outdated, being slower than Windows, not supporting the extra video features of the graphics cards. The development of these features was limited by the fact that XFree86 version 3.x had one video server for each major graphics card manufacturer. All the development was done inside the XFree86 project. The solution came in March 2000 when the version 4 of the XFree96 server was released. This new version had a single server, for all cards, and different video cards were supported through video drivers, loaded by the server. This allowed card manufacturers to provide binary modules for their cards, without the need to get involved in the whole XFree86 project. Another important improvement brought by the version 4 was the XVideo extension (XV). This extension is an API that enables the use of the hardware for color space conversion, filtering and scaling. This makes an application using the XVideo extension much faster with better picture quality, but old applications need to be adapted in order to use this new extension. All Linux players supporting XVideo can be considered as belonging to a new generation of players.

One of the first players to support XVideo (using Loki's SDL library [125]) was X Movie Player System (XMPS) [108], which wanted to become for video what XMMS [107] was for audio. XMPS was a plugin-based player, with new codecs supported by new plugins. XMPS can play MPEG-1, MPEG-2 Program Stream and several other codecs, but all development ceased in 2001.

Another emerging project at the beginning of year 2000 was GStreamer [103], which was intended to be a multimedia framework, mainly for Gnome. All new codecs and filters are implemented as plugins. A GStreamer-based application is a graph of media-handling components (plugins), similar with DirectShow.

Today, there are several successful video players under Linux and DVD playback has become easier too.

The XINE project [106] started in 2000 and it is currently one of the most used video players in the Linux world. It is a plugin-based video player with an impressive list of supported file formats, video and audio formats, output devices and other features (e.g supports subtitles). The GUI is separate from the main program which makes XINE suitable for many environments and for many types of user interfaces (including STB). The main philosophy in XINE development was to use readily available software, instead of creating everything from scratch. The result is a clean, stable code, but sometimes new codecs are supported later than other players. Currently, XINE supports DVD including menus, MPEG-2 TS, DivX, XviD [109], WM and other codecs, playback from different sources, including file, network (RTP, RTSP) and DVB cards.

The MPlayer project [104] started in November 2000. MPlayer is a plugin based video player that has also an impressive list of supported file formats, video and audio formats, output devices and other features (subtitles, image and audio post processing). Like in XINE, the GUI is separate from the main program. The philosophy in MPlayer is to let every hacker to do as he wants and hopefully something useful emerges. As a result, the code is ugly and less stable compared with XINE, but usually MPlayer is the first to support a new codec.

The Ogle project [105] started in late 1999 with a clearly defined scope: DVD playback. It was the first Linux player to support DVD menus. The menu support was later released as library and incorporated in other players also (e.g. XINE). Ogle has its GUI separated from the rest of the program, but does not seem to support anything else than DVD playback.

There are also other Linux players and many other projects started and abandoned over the last few years, leaving behind some useful pieces of code, many which are used in the video players today. An exhaustive history and analysis of the video players in Linux is outside the scope of this section. More details about the history of video and DVD playback in Linux can be found in [67].

### 4.3 Related scientific work

In the scientific world there are several papers dealing with media players and their architecture. In [135] the authors describe a mobile environment based on embedded Linux, capable of running several applications, including an MPEG-4 player, based on RTP. While the embedded system and the mobile framework, including the WLAN-based mobile IP are extensively described, no details are given about the player. In [71] the author presents an MPEG-2 software player (decoder) and makes extensive performance evaluation and implementation profiling. In [129] the authors briefly present a MPEG-4 system (Server and Client) based on Microsoft DirectShow. Architecturally, different modules (decoders, DMIF filters, Compositor) are implemented as DirectShow Filters. The architecture is modular, like any application based on DirectShow. The same system is described in more detail in [128] and sample MPEG-4 applications are presented. The DMIF instances on both client and server side are implemented as separate processes, which makes the whole system slower. The DMIF implementation uses RTP with an unspecified profile, TCP and plain UDP for the transport part. The signalling part uses TCP and it is not otherwise specified. The client is able to play MPEG-1/2/4 streams. In [115] a system that integrates hyperlinks to continuous media into the web is presented. The system is based on Java and scripting, with a multimedia player based on Java Media Framework (JMF) version 1 [93]. In [68] a modular MPEG-4 player is presented. Every module has its own role and the player can not be extended. Networking part is not clearly specified. In [60] a client server system for streaming content in homes is presented. The hardware aspect of the system is described in more detail. In [123] it is presented a hardware-based personal Digital Video Recorder (DVR) with modular software architecture running on dedicated hardware using its own API. In [15] it is presented a middleware system that introduces the concept of so-called Infopipes for data communication inside a system. Infopipes can be used in many types of systems, including multimedia. The creation of an MPEG-2 player using Infopipes is exemplified. Infopipes can be seen as a generalization of the Queue tool used by our player. However, we note that [15] was published in 2002, two years after our work.

## 4.4 The player architecture

### 4.4.1 General components description

The player architecture has three types of components: the core, the plugins and the tools (Fig. 4.1).

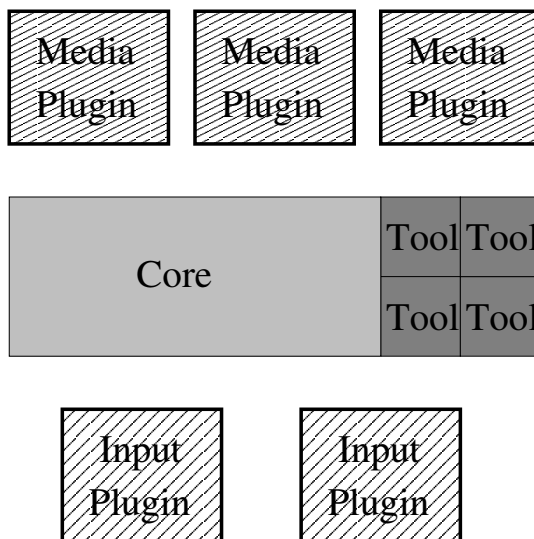


Figure 4.1: Simplified player architecture.

The core is the central unit of the player and it is responsible with keeping the state of the system and executing commands (like play or pause) that are send from outside. The core also commands the loading of some plugins, and informs them of the actual state of the system.

The tools are independent components, with their own APIs, linked together with the core in one file (the main executable). Their purpose is to provide services to the core or to the plugins. It was envisaged that each tool can have its own version, allowing its API to be modified slightly from one version to another. Such an API modification should require little change in the rest of the player (core and plugins using the tool).

The plugins are external dynamic-link libraries that reside in a directory known to the core. The needed plugins are loaded at runtime. We can split the plugins in two main categories, based on their role in the player: input plugins, interacting with the user and sending commands to the core, and media plugins, processing the multimedia data in the media chain, from acquisition, demultiplexing, decoding to output/rendering. The media plugins can also be split into the above-mentioned sub-categories, according to their role in the media processing chain. However, there is no fixed delimitation, an input plugin can also process data, a media plugin can also send commands to the core (when it detects the end of stream, for example) and a media plugin can have several functions in the media processing chain (such as

demultiplexing, decoding and rendering).

Fig. 4.1 illustrates the main concept of the player components: we see the tools linked together with the core into a single, unitary executable, but the tools still act as independent components providing services. The plugins are separate entities (files), only loaded when they are needed.

A more detailed player architecture is shown in Fig. 4.2. The core is split in few sub-components (objects) and the relations between components and sub-components are shown.

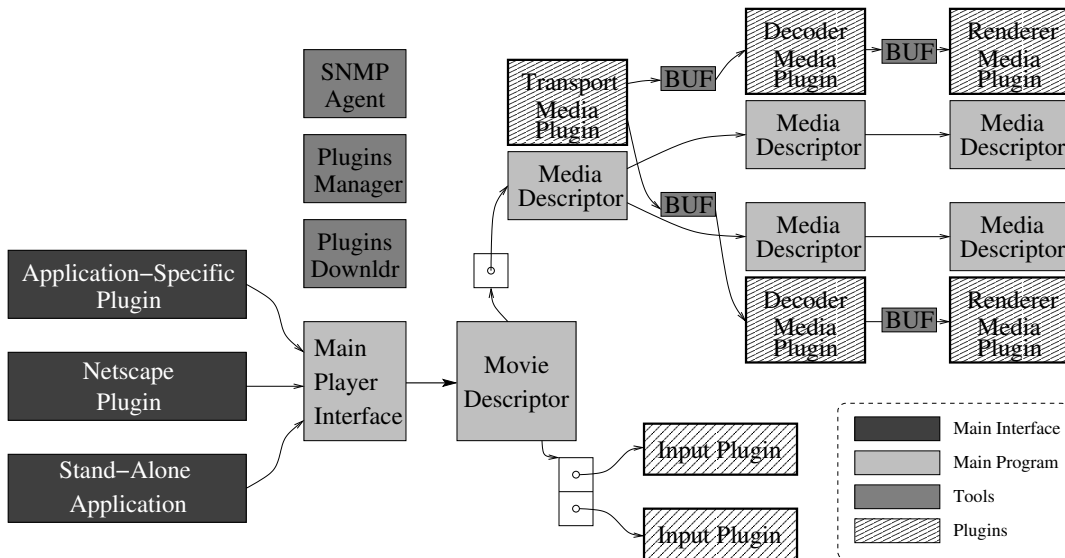


Figure 4.2: Detailed player architecture.

One of the main sub-components of the core is the *MovieDescriptor* object. It is implemented as a C++ class and contains information about the whole movie, both static (that does not change during the movie playback - description, duration, encoding type, geometry, number of frames per second, etc.) and dynamic information (that changes during movie playback - state of the system, playing position, viewing parameters, etc.). The *MovieDescriptor* object also contains functions for changing the dynamic information - the state of the system. After the state of the system has been changed by an input plugin or by other external means, the new state is made known to all directly registered plugins.

The *MainPlayerInterface* object is also implemented as a C++ class and is meant to act as an interface of the *MovieDescriptor* object to the outside world. It provides simple initialization and destruction methods and basic commands (**play**, **pause**, **stop**) that then translate to the more complicated interface to the *MovieDescriptor* object. The purpose of the *MainPlayerInterface* object is to ease the interconnection of the player with other software. There are several possibilities of interfacing the player: stand-alone, plugin for a web-browser or component of another application.

When implemented as a stand alone program, a main function has to be created, where a *MainPlayerInterface* object is created and initialized. Then it can be checked

if a filename or URI was provided when the player was launched, and the *MainPlayerInterface* object is asked to open it. Then the main function should wait until the program finishes. When implemented as a web-browser plugin, a similar approach can be followed.

The *MediaDescriptor* object contains information about the media processed by one media plugin (the URI of the content, the type of media, the sub-category of the plugin, references to the next *MediaDescriptor* in the processing chain, references to the data pipe objects between the current plugin and next plugins, a reference between the data pipe object coming from the previous plugin). There is one *MediaDescriptor* associated with each media plugin. Input plugins do not have a *MediaDescriptor* object associated with them. Each *MediaDescriptor* object is responsible for command propagation. It receives commands from the previous plugin or from the *MovieDescriptor* and sends the command to its associated plugin and to the next *MediaDescriptors*, that will do the same.

The player has several states: no stream, initializing, playing, paused, stopped, exiting. The transition between states is presented in Fig. 4.3.

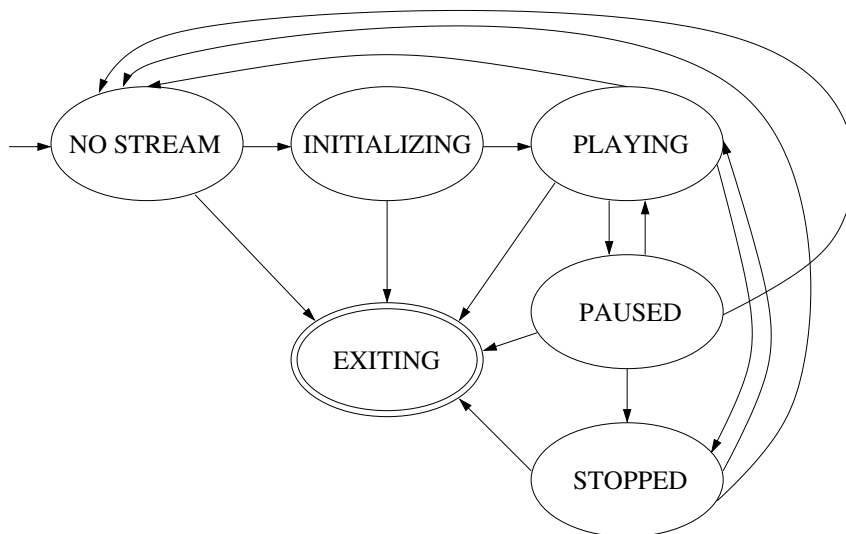


Figure 4.3: Player states and transitions between states.

When the player is initialized, it automatically enters the no stream state. This means that there is no open file or URI, and there are no media plugins loaded. However, all the input plugins are loaded (so that the user can open a file or URI, or close the program).

When a file or URI is opened, the player goes to the initializing state. Here the *MovieDescriptor* object tries to load a transport plugin able to handle that URI. For this purpose, a *MediaDescriptor* object is created and filled with the URI of the movie. The *MediaDescriptor* plugin type property is set to transport, and the best suitable plugin is loaded (using the *PluginsManager* tool). The loaded plugin tries to complete the media processing chain, if it is not able to do all the media processing (demultiplexing, decoding and rendering) by itself. To complete the media processing

chain, the loaded transport plugin creates and fills a *MediaDescriptor* object with the information about what plugin has to be loaded next, and passes this information to the *PluginsManager* tool, to load the plugin. This continues until the whole media processing chain is completed (as in Fig. 4.2).

When the processing chain is completed, the player enters the playing state. A **Play** command is automatically issued by the *MovieDescriptor* object after the initialization finishes. The command is sent first to the input plugins, in order to update their state, eventually. Then, the command is sent to the *MediaDescriptor* object of the first plugin loaded, the transport plugin. This object sends the command to its plugin, then propagates it to the next direct plugins, until the last plugins in the processing chain received the command.

The movie is now playing, and subsequent stop, play or pause commands may be issued at will. A command that will not change the state of the system is discarded (e.g. a **stop** command when the system has already been stopped). The system can not go from the stopped state to the pause state. The system has a current playing time parameter, so play commands changing the current playing position can be issued. The system has also a playing speed, that can be fractional and negative as well. A playing speed of one means normal playing, a fast forwarding effect can be obtained by playing the movie with increased speed (if the media plugins support this operation). A playing speed less then zero means reverse playing, that can be slow, normal, or fast, depending on the absolute value of the playing speed parameter.

The exiting state can be reached from any other state. This is also the final state of the player, it is not possible to go to other states from here. In this state, the plugins are stopped and unloaded. First, the input plugins are destroyed and then unloaded. Next, the exit command is issued to the first *MediaDescriptor* object, that corresponds to the transport plugin. This object stops its corresponding plugin, then issues the command to the next *MediaDescriptor* objects, that will do the same. When the function issuing the command to the next *MediaDescriptor* objects returns, the current *MediaDescriptor* destroys the next *MediaDescriptor* objects, destroys its associated plugin and then returns. At the end, the *MovieDescriptor* destroys the *MediaDescriptor* object associated with the first plugin (the transport plugin).

## 4.4.2 Plugins

A plugin for the player is a dynamic linking library that contains a class extending the generic plugin class, defined as an API. The plugin should also contain a C function that returns an instance of the plugin class, if the type of plugin corresponds to the type of plugin that it is wanted to be loaded.

The class should contains several member functions (the API of the plugin):

The *IsYourMedia* member function is called by the *PluginsManager* when it tries to load a specific plugin. The function has as input parameter a *MediaDescriptor* containing the information about the plugin type the *PluginsManager* intends to load. The function should return an integer that states the degree of suitability of the plugin with what it is desired to be loaded. A value of zero states that the plugin does not fit at all.

The *Initialize4Loading* member function should verify that all the conditions for the plugin to run properly are met (if the plugin depends on specific hardware to run, here is the place to verify that the hardware exists). If the running conditions are not met, the function should return a negative value.

The *Initialize4Running* member function initializes the plugin for use. When this function is called, it means that this plugin was already selected to run, from all the other available plugins. This function should initialize the current plugin and prepare everything for receiving data, processing it and sending it to next plugins.

The *DestroyPlugin* member function should prepare the plugin for unloading

The *OnCommandReceived* member function is called by the plugin's associated *MediaDescriptor* object to announce a system state change. The plugin should take whatever actions it finds appropriate.

The *OnDataReceived* member function is called when the plugin received data for processing, from its parent plugin. The function specifies a data field and an information field. The two plugins involved in the data transfer should have an agreement upon the format and significance of the information field.

The *SendData* member function is called internally by the plugin in order to send data to one of its child plugins. A generic implementation is provided by the generic plugin class. Data and information fields are specified, same as for the *OnDataReceived* function.

### 4.4.3 Tools

There are four tools considered in the current architecture, but more can be added if required by further developments.

One of the most important tools is the *PluginsManager*. This tool is responsible for loading one or all plugins that meet certain criteria. First, all the plugins that are found in the plugin directory are loaded. Then, the function returning the plugin class instance is called. The plugins for which this function returns NULL are discarded. For the remaining plugins, the *Initialize4Loading* is called, and all plugins for which this function fails, are discarded. Finally, *IsYourMedia* functions is called. If we have to load a single plugin, we select the one for which this function returned the maximum result, if we have to load all plugins that fit, we load all plugins for which this function returned a result greater than zero.

Another important plugin is the buffering data pipe between two plugins. This is implemented as a circular buffer with a producer (the sending plugin) and a consumer (the receiving plugin). Three possible consuming methods are defined:

1. A pull method, where the consumer explicitly calls a consume function that returns a buffering element, consisting of a data part and an information part. The consume function blocks if there is no data in the buffering system, until the producing plugin produces an element.
2. A push method, where a consumer callback linked to the *OnDataReceived* function is called when the buffering system is full.

3. A timestamp based method, where a consumer callback linked to the *On-DataReceived* function is called when the time registered in the timestamp associated with an element has come.

The buffering system allows several buffering instances to have common timestamp basis and delivery mechanism, to ensure proper synchronization when needed.

#### 4.4.4 Media playback with software decoding

In this scenario we analyze the plugins creation and command flow for a player receiving an RTSP URI of an MPEG-2 transport stream. Media decoding is performed entirely in software, and there are separate plugins for transport, demultiplexing, audio and video decoding, audio and video rendering (Fig. 4.4).

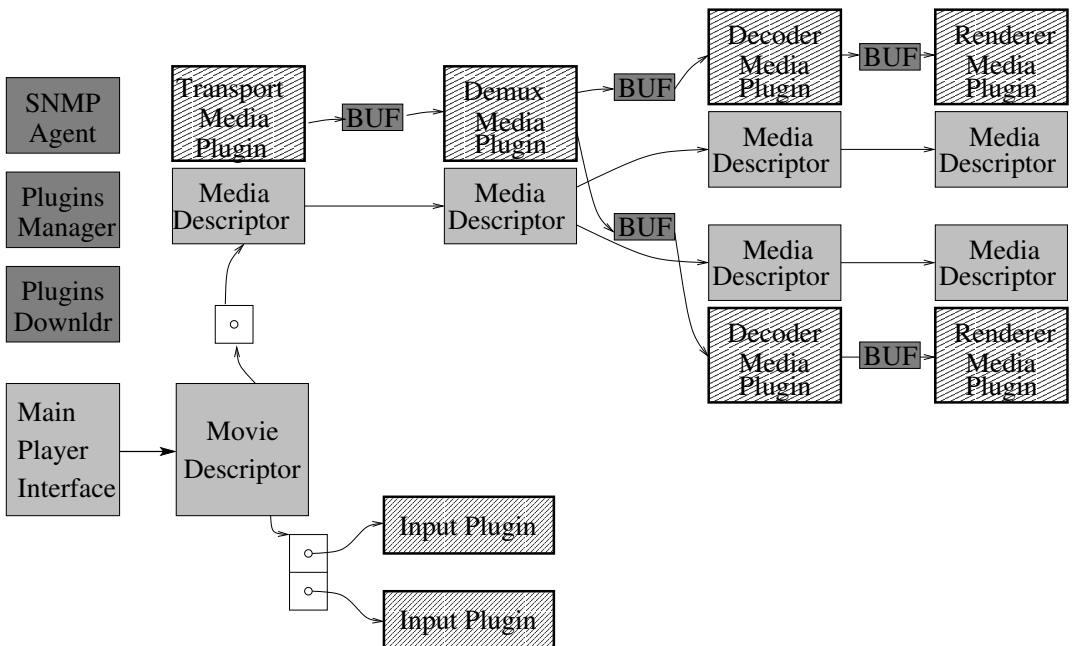


Figure 4.4: Media playback using software decoding plugins.

When the RTSP URI is received, the *MovieDescriptor* object is creating a *MediaDescriptor* object and fills it with information (URI and transport plugin type). The *PluginsManager* tool is called to load the best available transport plugin, that is capable of opening an RTSP URI. Suppose now there is only one such plugin (based on the joint RTSP, RTP and SDP library described in Chapter 2), and this plugin is loaded. The *Initialize&Running* function for this plugin will be called. Inside the function, the plugin starts the streaming of the content. It sends an RTSP **DESCRIBE** request to the server specified in the URI. If the response is OK and initialization data is received, the plugin may fill some of the static information fields from the *MovieDescriptor* object. Assume that the server is specifying in an SDP attachment to the **DESCRIBE** request that it accepts unicast connections for multiplexed

MPEG-2 transport stream, RTP-encapsulated. The plugin will issue a **SETUP** request for the specified stream. At this point the plugin will have information about the specific stream type which it will receive and that will have to be demultiplexed. So the transport plugin fills in a *MediaDescriptor* object for loading a demultiplexer plugin, capable of demultiplexing MPEG-2 (the media type is MPEG-2, plugin type is demultiplexing). Then the Initialize4Running function returns, and the associated *MediaDescriptor* function that propagates the initialization command will try to load a demultiplexing plugin. We suppose that there are two demultiplexing plugins available, but one needs specific hardware that is not available on this platform, so its Initialize4Loading function will return a negative value. The other demultiplexing plugin is loaded. We can assume that it is not able to decode the video and audio elementary streams, so it will also fill two *MediaDescriptor* objects, having the decoding plugin type, and audio respective video media types. After the Initialize4Running function of the demultiplexing plugin returns, its associate *MediaDescriptor* object will load the two decoding plugins. These two plugins will each load in the same manner corresponding audio and video rendering plugins.

When the whole chain finishes the initialization process, the initial function called by the *MovieDescriptor* object returns. Here the object will send the play command to the first *MediaDescriptor*, that will propagate it through the whole chain.

#### 4.4.5 Media playback with hardware-aided decoding

In this scenario we analyze the plugins creation and command flow for a player receiving a file URI of an MPEG-2 transport stream. The decoding is performed entirely in hardware, from demultiplexing to rendering, and there is one transport plugin that is able to read the stream from the file and a plugin that is able to initialize the hardware decoder and to feed it with an MPEG-2 transport stream (Fig. 4.5).

When the file URI is received, the *MovieDescriptor* object is creating a *MediaDescriptor* object and fills it with information (path and transport plugin type). The *PluginsManager* tool is called to load the best available transport plugin, that is capable of opening that file. We suppose there is only one such plugin, and it is loaded. The Initialize4Running function for this plugin will be called. Inside the function, the plugin opens the file, and it discovers an MPEG-2 transport stream. It then fills a *MediaDescriptor* object with information to load a MPEG-2 demultiplexing plugin. We suppose that the *PluginsManager* function finds two available demultiplexing plugins, one that uses the available hardware decoder and the other that is fully in software, and both are able to operate. The plugin that uses the hardware decoder should return a higher value of the IsYourMedia function in order to be loaded. This plugin is loaded and initialized. In its Initialize4Running function the hardware card should be initialized. This plugin has the functionality of the whole chain, so no further plugins are needed, so no further *MediaDescriptor* objects are initialized and the Initialize4Running function returns. After the play command is issued, the transport plugin will read data from file and send it to the demultiplexing plugin, that will forward it to the hardware card.

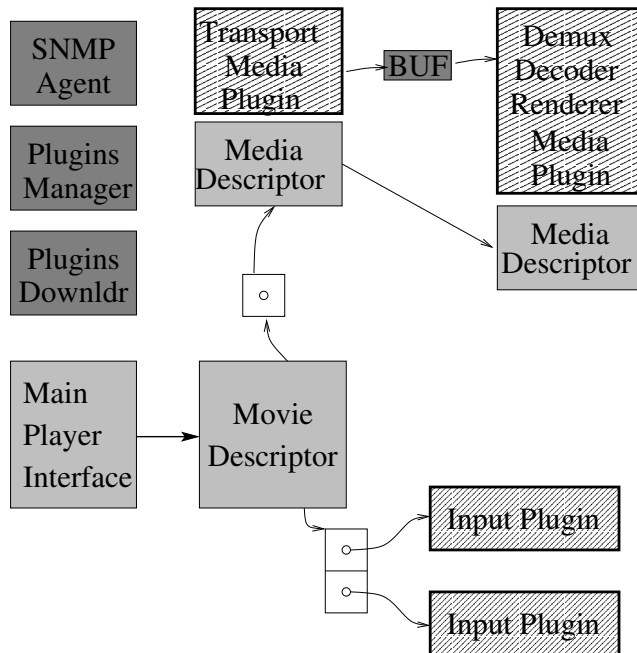


Figure 4.5: Media playback using hardware aided decoding plugin.

## 4.5 Summary and author's contributions

In this chapter we addressed the creation of a suitable video player for a STB and PC environment. Such a video player, capable of playing MPEG-2 and capable of network-based video retrieval was not existing when the presented player was started. The created player fulfills all its requirements by having a modular architecture and implementing all its media processing and control objects as plugins.

This player was one of the first player for Linux having a modular architecture and being capable of playing MPEG-2 TS from a remote server. This is also its main scientific contribution. This work or parts of it were presented in [86, 81, 82, 77, 76, 85].

The author designed the architecture of the player and implemented the core and the tools (except for the SNMP and Plugins Downloader tools). The plugins for this player were implemented or contributed by Prakash Sastry (MPEG-2 software video decoder, software demultiplexer and software rendering Plugins), Bogdan Iancu (GUI Input Plugin) and Diana Calugarescu (software audio Plugin).

## Chapter 5

# The Architecture of a VoD Server

In this chapter we study the problem of how to determine the maximum streaming performance of the storage subsystem in a VoD server. We also study how to establish a trade-off between necessary resources and the streaming performance in the storage subsystem. We create a behavioural model of the storage subsystem that, given some physical parameters of the storage subsystem and the target number of streams, computes the required resources to reach this target. We also design an algorithm (we call it Disk Pump) that manages the required resources and ensures that this target number on streams is achieved. We implement the Disk Pump algorithm in a simple test program in order to measure its performance and to validate our behavioural model. In the end, we design a VoD server based on the Disk Pump algorithm and on the communication library presented in Chapter 2.

We demonstrate that, using our Disk Pump algorithm, it is possible to stream 4 Mbits/s MPEG-2 TS streams to up to 50 clients, using a PC with a storage subsystem made of a single ATA hard-disk. We also demonstrate that it is possible to stream 4 Mbits/s MPEG-2 TS streams to up to 100 clients, using a software or hardware ATA RAID made of four hard-disks.

### 5.1 Introduction

Recent increases in the available bandwidth to residential Internet users (xDSL, Cable Modem) and advances in the consumer Set-Top-Boxes (STB) bring the promises of the Video on Demand technology closer to reality. Customers with high-bandwidth links to their homes (3-8 Mbits/s) and STB capable of Internet access and decoding the multimedia streams (MPEG-2 in the first phase) will be able to enjoy entertainment content (movies) at any time, on their TV. For this technology to have commercial success it is required that watching a movie using VoD or renting it (e.g. on DVD disk) to have a similar price. This puts hard price limits to the customer's STB, to cost of the data transfer and also to the cost of the VoD server.

Taken into consideration the recent advances in PC storage and networking technologies it makes sense to design VoD servers based on common PC hardware (ATA hard-disks, Gigabit Ethernet). Such servers may prove themselves to have the same

high-end features as servers based on more expensive technologies (SCSI, ATM) but at a much lower price.

## 5.2 Designing VoD servers

To meet the challenges and objectives of a VoD system deployment, the multimedia servers need to satisfy many requirements. These requirements are imposed not only by the VoD application itself, but also by several business considerations: cost and deployment [122]. By cost requirements we refer to issues making a VoD system cost-effective. By deployment requirements we refer to issues that make the operation of a VoD system cost-effective in the long run.

A short overview of these requirements is given below:

### 5.2.1 Application requirements:

- **Time sensitivity:** VoD streams are sent and received in real time. This means that the acceptable packet delay and jitter at the client side are limited. This limit, however, is not as strict as in the VoIP or IP conference cases, where the delay between the sampling of the baseband audio-video signal at the sender end and the audio-video rendering at the receiving end should be at most hundreds of milliseconds. In a VoD application, the delay between the sending of packets and rendering of the audio-video information contained in those packets may be of the order of seconds, depending on the client buffering capabilities and the client architecture (e.g. if the VoD system uses packet retransmission as an Error Correction method).
- **Open interfaces:** Different components of a VoD system need to be interoperable. For this, they need to have the same (or interoperable) transport and control protocols.
- **Client, network and server capabilities:** In a simple VoD system, the client should be able to start, stop and pause a stream. The server should be able to support these commands and it should be able to handle a reasonable number of commands per second (while it is quite improbable that a server with a streaming capacity of 100 simultaneous streams will receive all these 100 requests in the same time, processing 10 requests per second should be possible). When the complexity of the application increases, different playing modes (e.g. fast forward, fast reverse), complex data retrieval protocols, QoS, or complex application operations (e.g. subtitles retrieval and display, differentiated levels of advertising and billing) may need to be supported.

### 5.2.2 Cost requirements

- **Cost performance:** In large-scale systems, small increases in efficiency may lead to large reductions in costs. This may include caching or nVoD techniques for popular content.

- **Standard components:** The functional components making a VoD system should be, as much as possible, off-the-shelf hardware or software components, or have standard, open interfaces. An off-the-shelf component can be replaced easier than a proprietary one (e.g. common PC hardware versus proprietary hardware). If a VoD server uses open, standard interfaces, it can be easily interconnected with other tools and utilities inside the system (e.g. if the server has a filesystem interface to its stored content, the content can be backed-up using standard backup programs).

### 5.2.3 Deployment requirements

- **Scalability:** The demand of VoD content may increase in time, so the VoD solution should be scalable. There are various optimizations possible when the number of clients of a VoD system is very large (of the order of hundreds of thousands of clients simultaneously), including aggregating client requests.
- **Reliability:** Failure in a VoD system may lead to large loss of revenue and company goodwill. It is highly desirable to ensure redundancy by using redundant components or architectures.
- **Dynamic adaptation to workload:** In a VoD system, predicting the most popular content or the variation in the demand for the service may be difficult and tricky. Policies that allow the VoD server to adapt to varying loads may be required.

### 5.2.4 Delivery in VoD systems

There is significant scientific literature concerning the delivery of multimedia streams to clients. The simple approach in VoD is to send a dedicated stream to each client. The advantage of this approach is its simplicity, it potentially supports operations like seek, fast-forward and fast-reverse, the resources involved are easily accountable and it does not require the client to have extra storage space or increased available bandwidth capabilities.

The main purpose of the work concerning delivery in VoD systems is to save resources on the server and network side. This can be done using several methods, called stream merging techniques, or aggregating client requests. All the stream merging techniques suppose that the VoD system supports a large number of clients, and requests concerning the same contents in a short period of time are likely to happen. Stream merging techniques fall into one of these two categories:

1. VoD-centric methods: The client gets the requested stream immediately. It is totally transparent for the user if the system merges its request with other requests.
2. nVoD-centric methods: The client has to wait for a certain period of time before the streaming starts. Methods falling into this category try to find a balance

between the waiting time, available network resources (bandwidth to the client) and client capabilities (local storage and available receiving bandwidth).

The most notable difference between these two categories is the main area of applicability: the VoD-centric methods are mostly useful in a networking environment, while the nVoD-centric methods are more suitable for broadcast environment.

Some of the stream merging techniques require that the client has large storage space available, able to store important parts of the streamed content, and additional receiving bandwidth. These methods generally use the extra client bandwidth to stream future parts of the content. At any time during the playback, the content is either retrieved in real time from the server, or it is stored on the local storage.

We can thus summarize and classify stream merging techniques into 4 main categories, combining the VoD, nVoD approaches with the need for local storage and additional bandwidth on the client side:

1. VoD-centric, no need for local storage or additional receiving bandwidth:
  - Adaptive Piggybacking [36, 2]: The main idea is to alter the media playback rate of close requests so that they can be merged. The stream requested first will be played slower while the stream requested later will be played faster. After a while, the two requests will have the same playing time, so they can both be sent the same stream (by using multicast for example). Significant variations in playback (e.g. more than 5%) are not possible, since they would cause perceivable media changes.
2. VoD-centric, require local storage and additional receiving bandwidth:
  - Patching [32, 56, 18]: Clients buffer later portions of the media file by joining an ongoing multicast stream, while receiving, at a higher speed, a unicast patch stream from the server for the portion of data that it has missed.
  - Hierarchical Multicast stream merging [27, 28]: This approach is similar with the Patching method, the main difference is that a patch can also be shared, after some time, by several clients.
3. nVoD-centric, no need for local storage or additional receiving bandwidth:
  - Simple approach (or Staggered Broadcasting [23, 25]): Entire copies or a content are repeatedly broadcast on separate channels, each at the media playback rate. The starting times of each broadcast are staggered evenly across the channels.
  - Batching approach [25, 26]: Collect a group of request and send them with one channel. Batching techniques differ by the algorithm used to establish when to serve requests and by the the limit imposed to the maximum time a client has to wait until his request is served.

4. nVoD-centric, require local storage and additional receiving bandwidth:

- **Periodic broadcast:** uses several channels to broadcast a single piece of content. Each channel broadcasts a different part of the content. The receiver should be able to receive from all the channels simultaneously and store (buffer) those parts that will be played in the future. Periodic broadcast schemes can be split into several categories:
  - Pyramid [132]: can reduce the maximum waiting time experienced by users exponentially with respect to the number of channels.
  - Permutation-based pyramid [1], skyscraper [57], greedy disk-conserving [31]: these schemes try to improve the pyramid scheme by addressing the buffering required at the client side.
  - Fast broadcasting [62, 63], Pagoda [98, 99]: these schemes try to improve the pyramid scheme by reducing the waiting time experienced by users after their request.

Extensive reviews of the stream merging techniques and also other issues related to the delivery in VoD systems can be found in [137, 89].

### 5.2.5 Disk retrieval in VoD servers

The storage subsystem in a VoD server is one of the most critical resources. This is because random access to data stored in a hard-disk brings a time penalty that decreases the overall performance of the data retrieval. The time penalty is due to the mechanical parts in a hard disk: to read a different location, several milliseconds are spent to seek the hard-disk's head to the new location. During this seek time, there is no reading of data. The optimization of the data retrieval in a storage system is equivalent to reducing the number of disk seek operations and shortening their time.

Optimizing the data retrieval has several aspects:

1. **Placement of data on disks** refers to how the data is organized in a filesystem, the block size of a data unit and how the filesystem meta-data stores the indexes of the blocks composing a file [72]. It is recognized that traditional filesystems are not optimal for storing multimedia data. This is because normal filesystems are optimized to store many small files. The unit block size in a normal filesystem is 4 KB. To support also big files, the meta-data is organized in a hierarchical fashion [126].

Multimedia data is stored as big files. To optimize a filesystem for multimedia files, the block size should be big (of the order of 256 KB, 1 MB) and the meta-data should be organized accordingly (e.g. linear). Work describing a multimedia filesystem is described in [53]. Work was also dedicated to create block placement policies optimized for VoD:

- **traditional block placement** policies use knowledge of file access patterns to place blocks as to minimize the seek overhead and hence, the response time [122, 34]

- **constrained-placement** policies attempt to explicitly bound the seek distance between successive disk blocks [122, 33]
2. **Disk retrieval** refers to how the data is read from the disk. There are two categories of disk retrieval policies:
- **round-based** policies schedule I/O requests only during periodic scheduling rounds. These policies typically assume that the playback rate is constant and attempt to guarantee continuous delivery deterministically. The data read each round is proportional to the stream's playback rate. Because the filesystem usually has fixed block sizes, this may not be optimal. Round-based policies may need to map the data they have to read each round to an integer number of blocks. Examples of such policies are Round-robin [122, 88], Scan [122, 127] and Group sweeping scheduling [122, 138].
  - **fixed block size** policies can schedule I/O requests aperiodically. The number of I/O requests for a stream per unit of time is proportional to the playback rate of the stream. An example of such policy is Scan-EDF (Earliest Deadline First) [122, 111, 112].

An important aspect of a storage subsystem in a VoD server is its capacity, both in term of data space and throughput. Most of the time a VoD server requires throughput and space beyond the capacity of a single disk. A solution to this problem is provided by Redundant Arrays of Inexpensive Disks (RAID). The main idea in RAID systems optimized for speed is to strip the data across several disks (Fig. 5.1 shows a RAID consisting of 4 disks). A file is striped in relatively small block sizes (32 KB, 64 KB, 128 KB) and each block is stored on one of the disks (F1 to F12 in Fig. 5.1).

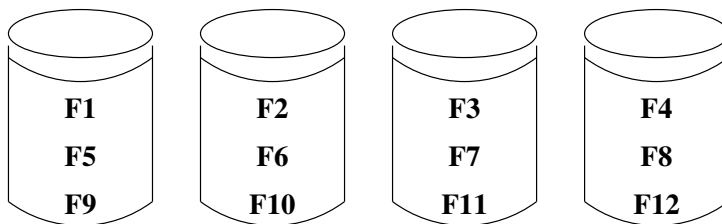


Figure 5.1: Striping a file across a RAID0.

Best results are obtained for I/O operations for blocks larger than the strip block size multiplied by the number of hard-disks. Using the RAID in Fig. 5.1, best results would be obtained for example by reading in a single operation F1, F2, F3 and F4, or F7, F8, F9 and F10 or F4, F5, F6, F7, F8, F9, F10 and F11.

There are several types of RAID, most used are: RAID0 – optimized for I/O speed, RAID1 – used to increase the reliability of a system, RAID5 – used for in-

creasing the read speed but still maintaining a good system reliability, and RAID10 – a combination of RAID1 and RAID0, used to provide both speed and reliability.

### 5.3 Behavioural model of the storage subsystem

In a broadband streaming server, one of the important problems we have to solve is maximizing the disk throughput considering that many concurrent read/write file operations may occur. Leaving this job to operating system and file system will produce a low throughput because normal file systems and operating systems are not optimized for large scale media streaming. It is known that any drop in performance from the sequential access throughput in an hard disk (30 to 50 MBytes/s for the read operation) is due to the seeks that take place in normal operation. During the seek time there is no throughput. In order to maximize the disk throughput during operation, we must either reduce the number of seeks (per second), or reduce the time required by a group of seeks, or both.

Beyond the simple file read/write operation there are three basic low-level operations:

- Seek the cylinder's head to desired position, the duration of this operation is dependent on the initial and final head position and disk's hardware characteristics. For current disks the full track time is approximately 14-18ms, the average seek time is approximately 8-12ms and the track-to-track time is approximately 1-2ms
- Wait until the desired sector of the cylinder goes under the read/write head; the duration of this operation is called latency and it is influenced by rotational speed. In a 5400 rpm disk one disk revolution takes approximately 11ms and the average latency is approximately 5.5ms, for a 7200 rpm disk one disk revolution takes approximately 8.3ms and the average latency is approximately 4.2ms
- Basic read/write operation for the data located on the current cylinder; the duration of this operation is dependent on the cylinder's data density and rotational speed

For simplicity we will define the complete seek operation compound from seek operation and latency. We can conclude that for current disks the average complete seek operation is around 12.2-17.5ms; worse case complete seek operation is around 22.3-29ms. For our future computations we will use the average of 15ms for a complete seek operation, which can successfully approximate current disks' performances.

Each of these low-level operations pays some tribute in the global disk throughput and using a smart approach for concurrent file streaming is required. The approach we use is a round-based policy, based on limiting the number of complete seek operations per second (which means limiting the number of streams switch per second). Actual limiting of the theoretical number of seeks per second is done by setting a fixed amount of data for read/write operations for each stream, length which is in direct dependency on the stream's bandwidth.

### 5.3.1 The Disk Pump algorithm

The Disk Pump algorithm that we implemented consists of a worker thread that sequentially processes the streams and a number of circular buffers, one for each stream. Duration of one stream processing is fixed for each stream and depends on the stream bandwidth and the targeted maximum number of seeks per second. Processing one stream for read case means that the worker thread reads in advance the stream into the stream's buffer. When user read command occurs, the data will be retrieved from the stream's buffer. For write case, the user write command will cache the data into the stream's buffer following that in the moment the worker thread will process the respective stream the data will be actually written on the disk from the stream's buffer. This will force that huge amount of data will be transferred between successive seek operations, maximizing total throughput.

There are two trade-offs that have to be considered when fine-tuning the disk-pump for specific hardware and software configuration:

The first trade-off we must consider is the maximum theoretical number of seeks per second versus total disk throughput for concurrent streaming. For example using 10 seeks per second ( $15\text{ms} \cdot 10 = 150\text{ms}$ ) we have left 850ms from one second for data transfer, which means 85% throughput of a continuous file streaming. Using 15 seeks per seconds ( $15\text{ms} \cdot 15 = 225\text{ms}$ ) will result in a 77.5% throughput of a continuous file streaming.

The second trade-off we must take into account is choosing the right number of seeks per seconds according to the available memory. If we target 10 seeks per second so our maximum desired load is 100 streams at 4Mbits/s each, then each stream gets its own time-slot once ten seconds, which means we must have a buffer for each stream of  $10\text{sec} \times 4\text{Mbits/s} = 5\text{MBytes}$ , and a total of 500MB memory for all the streams. If we recompute for 15 seek per second, we found that the requirements are  $3.33\text{MBytes} \times 100 = 333\text{MB}$  of memory.

### 5.3.2 The behavioural model algorithm

The behavior of a storage subsystem can be modeled and we can compute the required number of seeks per second and the necessary memory resource for a given target performance. The behavioural model has the following input parameters:

- $B_c$  = The continuous read speed of the disk subsystem (in MBytes/s)
- $t_s$  = Average time of a complete seek operation (in milliseconds)
- $b$  = Average bitrate of a single stream (in Mbits/s)
- $N$  = Targeted number of streams
- $p$  = Overcapacity of the buffer for a single stream (in percents)

In order to have a solution, the input parameters have to fulfill the following condition:  $N \cdot b < B_c \cdot 8$ . This condition states that the aggregate bitrate of the streams can not be higher than the continuous read speed of the disk subsystem.

The model produces the following values:

- $n$  = Number of seeks per second
- $t_r$  = Time between two consecutive reads from disk for the same stream (in seconds)
- $M_{buf}$  = Required buffering space for one stream (in MBytes)
- $M_{tot}$  = Total required buffering space for all the stream (in MBytes)

The model computes the output parameters as follows:

The number of seeks per second:

$$n = \frac{B_c \cdot 8 - N \cdot b}{B_c \cdot 8 \cdot t_s / 1000} \quad (5.1)$$

The time between two consecutive reads from disk for the same stream:

$$t_r = \frac{N}{n} \quad (5.2)$$

Needed buffer for a single stream:

$$M_{buf} = t_r \cdot (b/8) \cdot (1 + p/100) \quad (5.3)$$

Total required buffering space:

$$M_{tot} = M_{buf} \cdot N \quad (5.4)$$

Example 1: We suppose the following input parameters:

- The continuous read speed of the storage subsystem:  $B_c = 75$  MBytes/s
- The average time of a complete seek operation:  $t_s = 15$  ms
- The average bitrate of a single stream:  $b = 3.75$  Mb/s
- The targeted number of streams:  $N = 100$
- Overcapacity of the buffer for a single stream:  $p = 20\%$

With these input parameters we compute the required number of seeks per second:

$$n = (75 \cdot 8 - 100 \cdot 3.75) / (75 \cdot 8 \cdot 15 / 1000) = 25.$$

We also get:

$$t_r = 100 / 25 = 4 \text{ seconds,}$$

$$M_{buf} = 4 \cdot (3.75 / 8) \cdot (1 + 20 / 100) = 2.25 \text{ MBytes,}$$

$$M_{tot} = 2.25 \cdot 100 = 225 \text{ MBytes.}$$

We can notice that for the above parameters the total disk subsystem throughput is  $100 \cdot 3.75 = 375$  Mb/s, which is  $375 / (75 \cdot 8) = 62.5\%$  from the overall throughput capacity. We can consider this 62.5% as the efficiency of our implementation parameters. Instead of targeting a number of streams and using the behavioural model to

compute the required number of seeks per second, we can start from the efficiency and compute how many streams the storage subsystem is able to handle:

$$N = \frac{\eta \cdot B_c \cdot 8}{b \cdot 100} \quad (5.5)$$

where  $\eta$  is the efficiency in percents. We can then compute the output parameters of the model. In the example above, if we choose an efficiency  $\eta= 90\%$  we can compute the targeted number of streams as:  $N = 90*75*8/(3.75*100)=144$ . We can then compute the required number of seeks per second:

$$n=(75*8-144*3.75)/(75*8*15/1000)=6.67.$$

We also get:

$$t_r=144/6.67=21.58 \text{ seconds,}$$

$$M_{buf}=21.58*(3.75/8)*(1+20/100)=12.14 \text{ MBytes,}$$

$$M_{tot}=12.14*144=1749 \text{ MBytes.}$$

We can see that high efficiency comes at a price (we would need almost 2 GB of memory for the buffering).

## 5.4 Performance evaluation of the Disk Pump algorithm

### 5.4.1 Test program and system

One important factor affecting all disk I/O operation in Linux is that all read/write operations (files, block devices) are cached through a kernel buffer cache mechanism. This cache mechanism uses memory from the free RAM memory in the system. To read for example 1 MByte of data from hard disk, this data is first read in the cache memory and then it is copied into its destination buffer in the application data space. This increases the performance of the usual applications that read small amounts of data at once from the hard disk, but it slows down uncommon applications that require high disk throughput and special file systems such as database servers and media streaming servers. Before the Linux kernel 2.4.17 one way to deal with this problem and avoid the cache system was to create raw devices (character devices associated with corresponding block devices) but this does not give the expected performance because of the implementation of the character devices that are not suited for high throughput and additional computations. Starting with kernel 2.4.17 a new method for avoiding the buffer cache is providing, by using the `O_DIRECT` flag for file descriptors. If a file descriptor of a block device or a file has the `O_DIRECT` flag set, then the read/write operations for that file descriptor are not going to use the cache system. If the file descriptor does not have the `O_DIRECT` flag set, then the read/write operations are performed through the system cache.

In order to test the performances of the Disk Pump algorithm and the influence of several factors to the overall disk I/O performance, we have prepared a test system with the following hardware and software configuration:

Hardware configuration:	Software configuration:
Intel Pentium III 800MHz	Linux Mandrake 8.1
1 GB RAM	kernel 2.4.17 (with O_DIRECT)
3ware 3W-6400 RAID Board	no swap partition !
4xIBM-DTLA-307030 30GB	no filesystem, block devices used !

The tests were made with single disk, 4-disk software RAID0 (64KB stripe size) and 4-disk hardware RAID0 (64KB stripe size) using normal access and direct access (with O\_DIRECT option) for the corresponding block device. Avoiding the filesystem ensures that the results of the tests do not depend on the partition fragmentation and on the filesystem performance itself (ext2, ext3, Reiser FS, XFS). Not having a swap partition ensures that no additional disk I/O operations are performed by the operating system during the tests.

A simple testing program was implemented for the special purpose of testing the performance and validity of the Disk Pump algorithm. In order to get away of any additional overtime (thread switching, thread synchronization) we designed a single threaded test program in C under Linux, that implements the basic functionalities of the worker thread from the Disk Pump algorithm. This testing program allocates two memory blocs. The first one is used to place the data that is read from the disk. Data is always placed in this buffer, which is several MBytes large (e.g. 5 MBytes). The second buffer is allocated but not used by the program after that. This buffer is used to limit the free memory available to the system. This free memory is used by the system for the cache mechanism. The testing program is able to perform tests on raw block devices and also on multiple files (through local file system); it is used for the fine-tuning of the disk pump module for a VoD Server [84, 85]. The test program tries to read as much as possible under the given configuration.

## 5.4.2 Test results

### 5.4.2.1 Test Set 1: continuous read speed

In these tests we want to determine the raw performance (one stream - continuous read) for three configurations (single disk, software RAID0 and hardware RAID0), using two different methods: normal disk access and direct access. The results of the tests are presented in Fig. 5.2. For each testing several block sizes (the amount read from the disk subsystem at once) are used: 64, 128, 256, 512 and 1024 KBytes.

Processing larger block sizes (the amount of data read at once) shows to be a real impediment in the buffer cache mechanism for higher throughput, and the maximum throughput cannot be achieved in this case. Direct access method looks more suitable for large amounts of streamed data. For the 64 KB block size, the amount of data read once corresponds to the RAID stripe, so only one disk is involved in this case. For the 128 KB case, two disks are involved, all disks start being involved only from 256 KB block sizes onwards. This explains the sharp increase in performance for RAID systems when the read block size increases. For the single disk case, the block size and the access method do not affect the performance results.

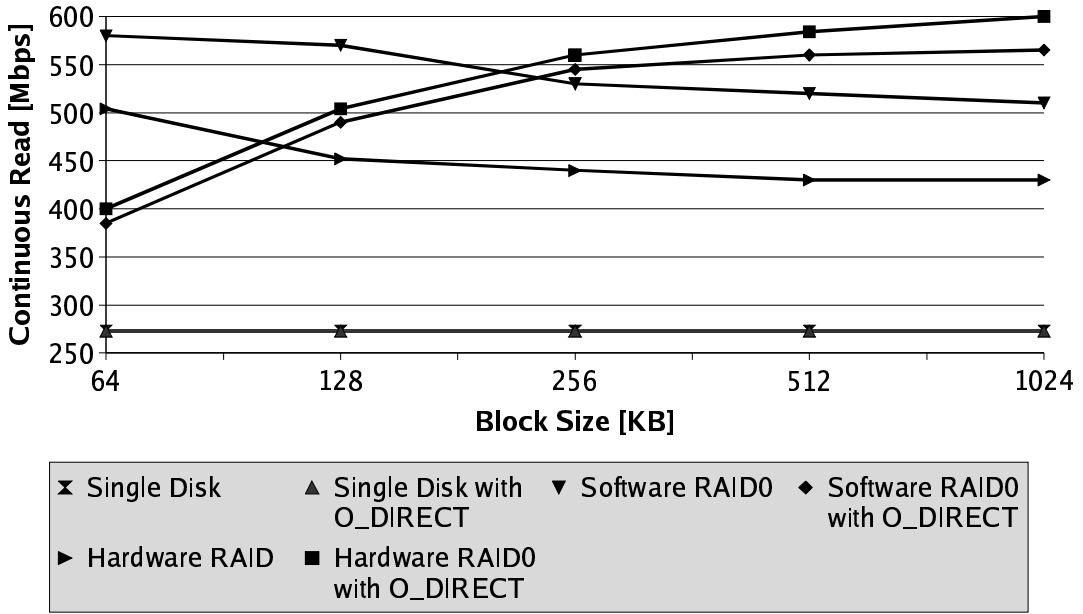


Figure 5.2: Results of continuous read.

#### 5.4.2.2 Test Set 2: Read speed when using Disk Pump - dependence of allocated memory

In these tests we aim for a target of 400Mbps/s total disk throughput using 100 clients, 4Mbps/s each, and limiting the number of seeks per second to 10. Moreover, we plan to see the performance changes regarding memory occupied, because this is an important issue in the real disk-pump module. In the real module, the necessary memory for buffering in this case will be  $100 \times 10 \times 4 \text{ Mbps} = 500 \text{ MBytes}$ . We are running the tests using first a 128 KB block size (results in Fig. 5.3) and then a 1024 KB block size (results in Fig. 5.4).

Using direct access shows to be an important issue in the disk pump. Tests show that the speed of the normal access starts decreasing when the test program allocates more than 300 MBytes. This shows that the system cache mechanism needs approximately 700 MB or RAM (1 GB - 300 MBytes) in order to perform at similar speeds as direct reading from the disk. The performance of the normal method further decreases when less and less memory is available for the cache mechanism.

We can also notice that the results obtained for single disk, and RAID with O\_DIRECT are around 85% from the corresponding continuous read results, which confirms our previous suppositions.

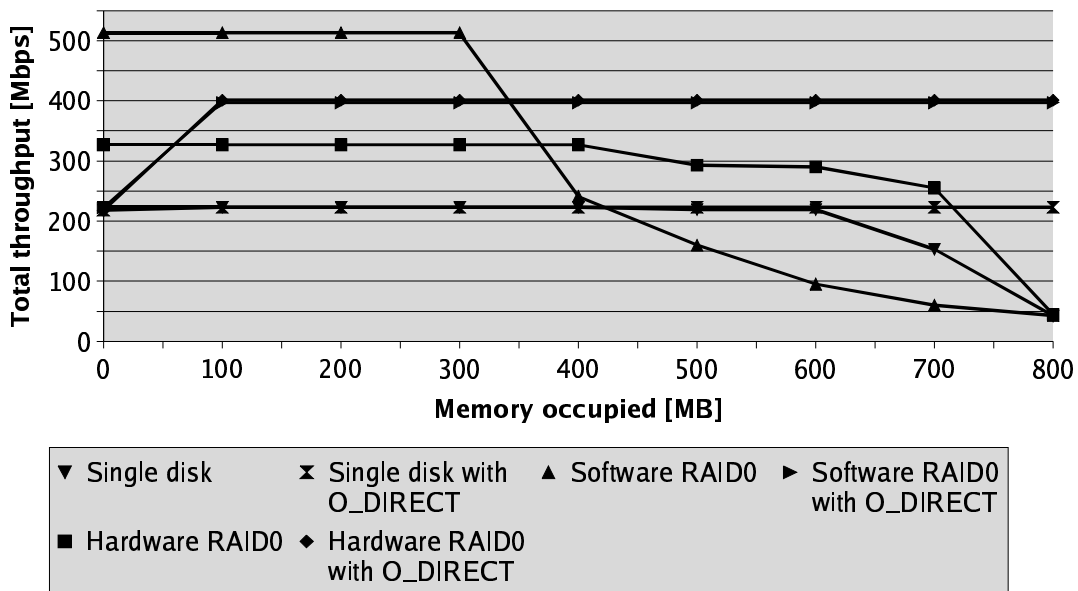


Figure 5.3: The influence of the cache system on the Disk Pump performance (target: 400 Mbits/s, 10 seeks/second, 128 KB block size) .

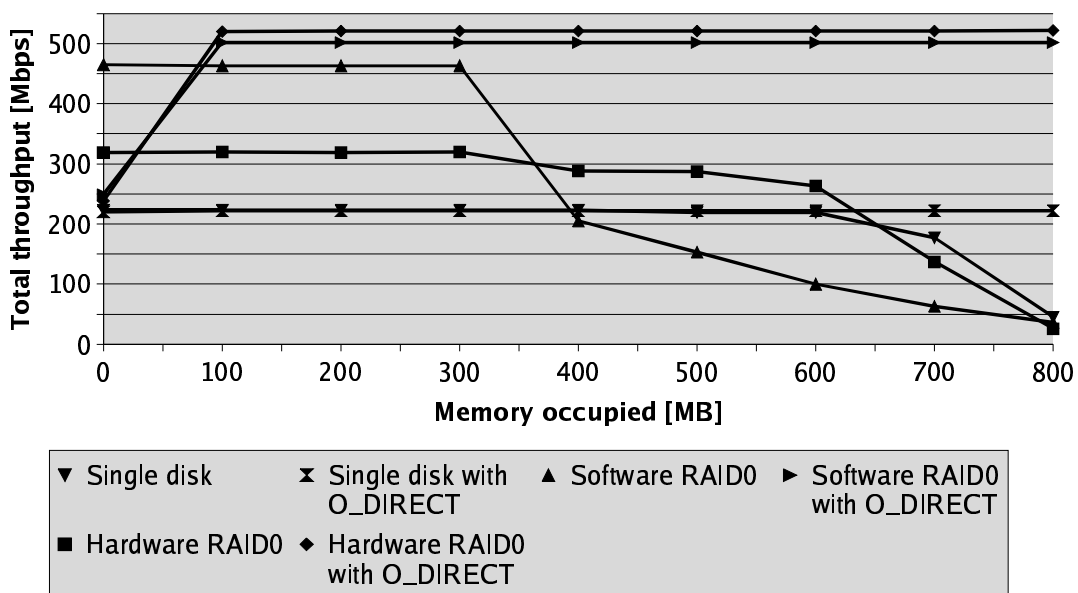


Figure 5.4: The influence of the cache system on the Disk Pump performance (target: 400 Mbits/s, 10 seeks/second, 1024 KB block size) .

The results for the 1024 KB block size, direct access method using both hardware and software RAID shows much better results than the 128 KB block size, for reasons previous explained (only 2 disks are used for one read in the 128 KB block size). The readings using the system cache in the 128 KB case are better, because the operating

system optimizes the actual reading from the RAID. The direct method for the 1024 KB block size performs better than the normal method, even if sufficient memory for the cache mechanisms is available. One possible reason for this is that the direct method avoids copying the data from one buffer in the memory (the cache) to another one (final destination of data in the application space).

### 5.4.2.3 Test Set 3: Read speed when using Disk Pump - Disk Pump speed, dependence on block size

In these tests we see the performance dependency using different block sizes for the hardware RAID0 in direct access mode. Also we want to conclude the difference between the computed theoretical disk-pump (85% of continuous throughput for 10 seeks per second) and the obtained results (Fig. 5.5).

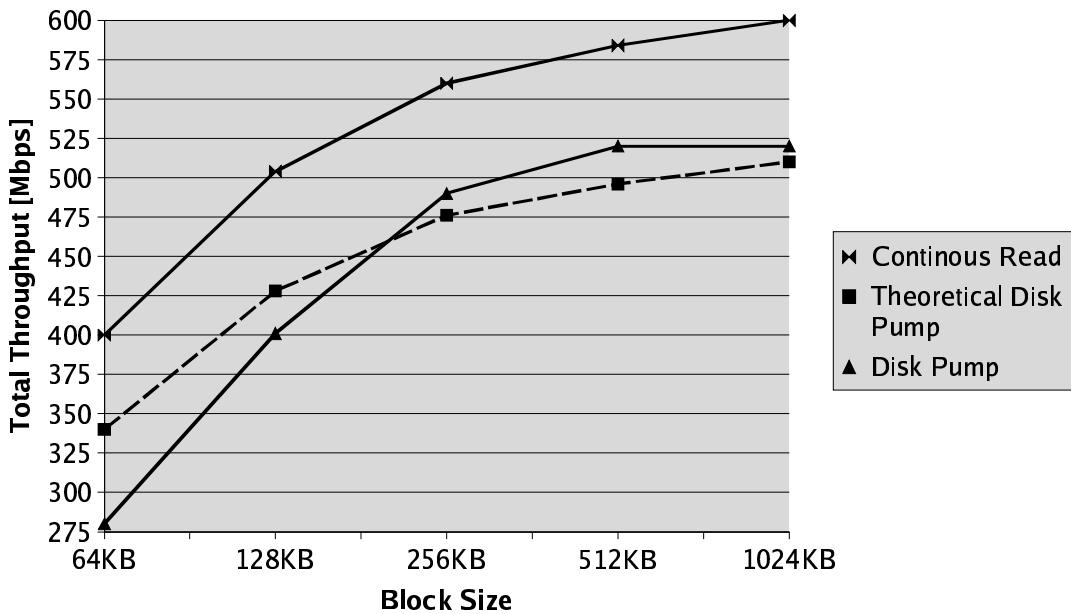


Figure 5.5: Total throughput of the Disk Pump algorithm - block size dependency.

Performance is greatly increased using larger block size because of the 64K stripe size of the RAID0 array. The theoretical and practical results are close, especially for large block sizes. This proves that our initial approach and the computations are valid. The small differences between the theoretical Disk Pump and our implementation are probably due to the way the operating system optimizes the seek operations and the read operations for small block sizes. In practice, it does not make much sense to read block sizes that are less than the strip size multiplied with the number of disks in the array (in our case: 64 KB x 4 = 256 KB). More than this, a multiple of this number should be used in practice, to avoid reading from some disks more than from others.

#### 5.4.2.4 Test Set 4: Disk Pump algorithm running inside the VoD Server

This test was conducted to check if the performance of the disk pump implementation inside the VoD server shows the same performance as the testing program. 100 clients were simulated to connect to the VoD server, each of them requesting a different 4 Mbits/s stream. The VoD was configured for 10 seeks per second. In this case each stream was having its turn to read from disk every 10 second, so  $4\text{Mbits} \times 10\text{seconds} = 5\text{MBytes}$  buffering was necessary for each stream, giving a total of 500 MBytes. The conclusion of the test was that the disk pump in the VoD was capable to serve all 100 clients and it was displaying same behavior as the testing program.

### 5.5 The architecture of a VoD server

A real VoD server was designed and implemented using the Disk Pump algorithm and the joint RTSP, SDP and RTP library presented in Chapter 2. Our VoD server is aimed at small systems, capable of streaming up to hundreds of streams simultaneously. In such systems, the probability that several clients request the same content in a small time window is very low, so possible resource savings by utilizing one of the stream aggregation methods described in section 5.2.4 are very low also. Because of this reason and for simplicity, the server sends one stream to each client.

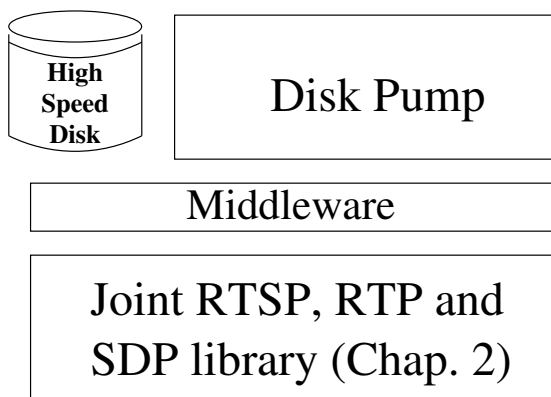


Figure 5.6: The architecture of the VoD server.

The server is capable of streaming MPEG-2 TS streams only. This greatly simplifies the problem of retrieval from disk and RTP encapsulation, because a TS stream is made of 188 bytes TS packets. Reading a TS from disk means reading a file continuously, at the playback speed. No demultiplexing is necessary (as would be the case with streaming from an mp4 file [39, 12]). Encapsulating an MPEG-2 TS into RTP [54] is easy and requires that each RTP packet contains an integer number of TS packets. The server architecture would also allow ATM to be used as a transport/control networking protocol. Tests were also made to check ATM capabilities as

a protocol for carrying multimedia streams [75]. Current PC architecture is capable of streaming more than 500 Mbits/s using Gigabit Ethernet.

The architecture of the server is presented in Fig. 5.6. The Disk Pump and the joint RTSP, RTP and SDP library are the main components of the server. Between them there is a thin middleware layer that translates the commands received from the client, through the communication library, to the disk pump.

## 5.6 Summary and author's contributions

In this chapter we addressed performance issues related to content retrieval from the storage subsystem in a VoD server. We developed a behavioural mode for the storage subsystem in a VoD server. This model, given some physical parameters of the storage subsystem and the target number of streams, computes the required resources to reach this target. We also implemented an algorithm which, using the parameters computed with the described model, is capable of managing disk resources and able to ensure the streaming of the targeted number of streams.

The scientific contribution of this work consists in designing the Disk Pump algorithm and demonstrating that it is possible to achieve the high performance required by a VoD server when reading multimedia streams with a PC running Linux and having an ATA RAID0. Our maximum achieved performance was 500 Mbits/s, when using hardware RAID0, direct access method and a block size of 1024 KB, the number of disk seeks per second was 10. This speed is enough to serve more than 100 clients requesting MPEG-2 streams (at 4 Mbits/s). The tests show that global system performance is highly correlated with the hardware and software configuration, and a correct tuning process is required in order to achieve the optimum performance.

The work presented in this chapter is based on [85, 87]. The author of this thesis contributed to this work by designing the disk pump algorithm and by making its initial implementation. The help of Marius Vlad with the implementation and performance evaluation tests on different platforms is highly appreciated.

## Chapter 6

# Management Platform for Multimedia System

Managing large-scale multimedia systems is one of the key problems in the multimedia industry nowadays. This is because such systems are usually composed of many heterogeneous devices, from different manufacturers, where each device is usually managed independently and comes with its own management system. Managing costs make an important part of the total cost of ownership in almost any system today, so optimizing management costs makes a lot of sense. Another problem in multimedia systems with end clients is the user interface to the whole system. Clients need to have all the available content in a tempting, easy-to-browse and easy-to-access way, otherwise they will refrain to watch (buy) content from that system.

The work presented in this chapter addresses both system management and user interface problems mentioned above. We describe the architecture of a platform for managing multimedia servers considering these two main goals:

- To allow a system administrator to configure, monitor and control a group of heterogeneous multimedia servers.
- To act as a portal/gateway for clients by presenting them personalized lists with the content available from the servers.

The platform is web-based and its implementation currently supports two services: VoD and IP TV. The IP TV service allows TV channels broadcast over broadband IP links. The IP TV service requires two types of servers: IP TV servers, which receive the digital TV channels and send them over the network, and Transcoding servers (Transcoders), which reduce the necessary bandwidth for each TV channel in order to fit the bandwidth currently available over the broadband links. Our management platforms supports these three types of servers: IP TV servers, VoD servers and MPEG-2 Transcoders.

In this chapter we give details about the initial requirements, how the platform was designed to fulfill these requirements, and about the implementation aspects.

## 6.1 Related work

The current large scale multimedia systems are usually composed of devices of heterogeneous nature, sometimes from different manufacturers. Managing such a system in an integrated manner is not a trivial task, but rather a key issue that should be considered at the design phase. The primary goal is to avoid that each device is managed independently and to create, whenever possible, a single point for system management. Currently, there are several solutions for this. One powerful general-purpose management protocol is SNMP [19]. On top of it, integrated and customized management solutions can be built. Other solutions involving newer management models are based on distributed object approaches (such as CORBA [44]), management by delegation [35], cooperative models with the use of intelligent agents [139]. In addition to these models, a new management solution has appeared as the result of the growing popularity of the World Wide Web: web-based management. This method has several potentially big advantages:

- Better accessibility. Because the web-based management uses a web browser instead of the classical management consoles, any computer with a web browser can be an access console.
- Platform independent. Any platform/operating system having a web browser can potentially be a management console. In classical approaches, the management application is only available on one or few (sometimes expensive) platforms.
- Simultaneous access to the managed system by several administrators from different places.

An exhaustive classification of web-based management systems is given in [131]. We briefly summarize it here:

1. Embedded/Direct approach. This architecture allows the management of each device individually. The device has its own web server and its IP address. While this method is simple and straightforward, a system composed of many devices becomes difficult to manage.
2. Proxy approach. This architecture aims at simplifying the management of the system by hiding the complexities of standard management protocols (SNMP). The key component in this architecture is the proxy, that translates the commands and information from and to the web world to the SNMP world.
3. Gateway approach. This architecture, similar with the Proxy approach, also involves a key component, the gateway that intermediates between the web world and SNMP world. In addition to the Proxy model, the client (web browser) and the gateway may communicate not only through HTTP but also through other TCP-based protocols. This is possible if a Java Applet is running on the client side, providing additional functionality. The Gateway approach has

a higher level of abstraction than the Proxy approach, where the commands mostly resemble their SNMP counterparts.

4. Mobile code - intelligent agent approach. This architecture combines intelligent agents and client software both based on a mobile code language such as Java. Autonomous modules - the agents - are executed remotely, closed to the system devices, under the control of the web browser/Java Applet.

The management platform presented in this chapter does not belong fully to any of the categories mentioned above. However, the architecture of the presented management platform partially resembles the proxy approach, because there is a central proxy that communicates with the administrator's web browser through HTTP. What is different is the communication between the proxy and the servers, which does not resemble SNMP. In our approach a new element is introduced, the database, and there is extra communication between the proxy and the database and between the database and servers. The presented management platform has also elements from the Gateway approach (because of a Java applet running on the client side), but this applet communicates directly with the servers and not with the gateway.

## 6.2 System components and initial requirements

The initial concept of the management platform was born from the need of configuring, controlling and monitoring two types of servers: Video-on-Demand (VoD) servers and IP TV servers. Later, a third type of server was added: MPEG-2 Transcoder. The system should be able to handle several servers of each type (of the order of tens) in a centralized and integrated way. The system should be also highly reliable. The choice of having a web-based management system came naturally, because this is arguably the best choice for an application that has to be accessed from several different operating systems. One may argue that a Java application or applet would make also a good choice, but there are issues related with running Java applications on different platforms, and also Java is not as widespread as a web browser. This is also the reason why we tried to avoid as much as possible Java applets in our web-based management platform.

The components of the system, including the management server are shown in Fig. 6.1. The management server communicates with the administrator through web (HTTP and HTML). The administrator sends commands and configures devices through this interface (as HTML Forms) and receives monitoring information in the form of HTML pages. Because of the potentially high volume of the configuration data required by the servers, a database is used.

The Management Server can set a two-way communication channel with each server. It can send commands to servers and servers respond with requested information or their status. Each server can, when needed, reads its configuration data from the database.

Few details about the types of servers used in the system are given below, in order to understand better their configuration needs.

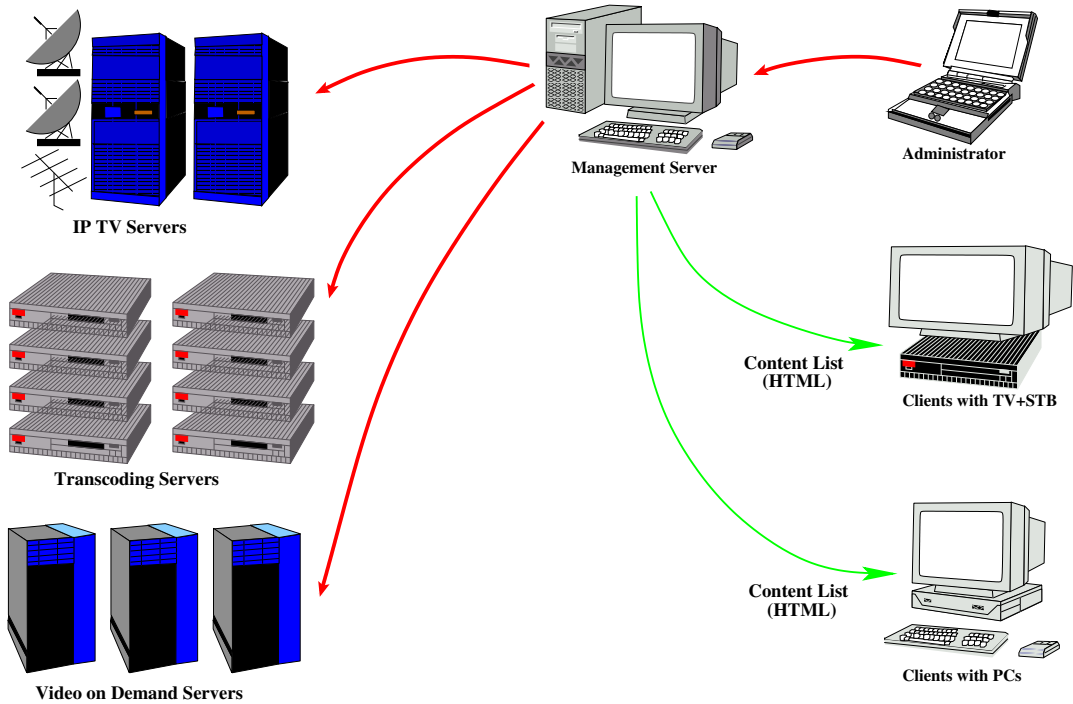


Figure 6.1: Components of the multimedia system.

### 6.2.1 VoD servers

These servers store streaming content (movies) on their local storage subsystem and stream it when requested by clients. For each server, specific parameters can be configured: RTSP port, maximum number of clients, available storage space for movies.

A movie in the system can be stored on several servers. Because of this reason, details about the movie (title, short description, language, rating, genre, cast, etc) are separated from the information specifying on what servers and in what format the movie resides. Thus, for each server storing the movie as a file (MPEG-2 Transport Stream), the management system keeps details about the filename on disk, file format, audio and video tracks inside the file container, the audio and video quality of the compression.

### 6.2.2 IP TV servers

These servers receive MPEG-2 compliant Transport Stream digital TV streams from a DVB source (currently DVB-S and DVB-T are supported) and they multicast these streams over the local network. The servers are based on Linux PC systems. Each server contains up to six DVB receiver cards. A DVB card can receive a satellite transponder (in the case of DVB-S) or a terrestrial TV channel (in the case of DVB-T). Each of these may contain several TV programs.

Configuration of DVB-S and DVB-T cards differs significantly. For a DVB-S

card to receive TV channels from satellite, a cable coming from a satellite dish has to be connected to that card. The server then needs to know details about this cable: the satellite (and beam) the dish antenna is receiving, LNB parameters, the polarization and frequency constraints of that cable, in case it is shared by several cards and they all have to lock it on same polarization (Horizontal or Vertical) and frequency interval (High or Low). Then, the server needs to know the parameters for the selected transponder and the Packet Identifiers (PIDs) of the digital TV programs to receive (in the case that we do not want to receive all the PIDs in that transponder). The received programs are then multicasted on specified IP addresses. Names, broadcasting languages and details can be associated with the multicasted programs. For a DVB-T card, the configuration is much simpler: the administrator has to choose the region where the DVB-T antenna (the server) is located. Then the administrator can choose which of the available programs the card should receive. Even if one card can only receive programs from a single digital TV channel, a single antenna and cable may be shared by several cards receiving different channels. In Finland there are 3 digital TV channels broadcasted, carrying a total of 12 TV programs, so to receive them all, 3 DVB-T cards are required.

The configuration of the servers is done in a user-friendly way, where the administrator chooses the satellites, cables, programs and regions (as in Fig. 6.3 and Fig. 6.4), rather than inputting the raw parameters needed by the cards (to receive correctly the desired TV programs). The database required for configuring the DVB over IP servers is fairly complex.

The IP TV Servers are presented in details in [85, 101].

### 6.2.3 Transcoding servers

These servers change the bitrate and possibly the geometry of an MPEG-2 Transport Stream (TS) in real time. The output format is also MPEG-2 TS. In our system, transcoders are usually used together with IP TV servers to reduce the bitrate of a broadcasted TV program. In this case, a transcoding server receives the multicasted MPEG-2 TS in real-time, transcodes it and resends the transcoded MPEG-2, also in real-time.

In addition to the input and output URI, the configuration of a transcoder involves basic and advanced parameters. As basic parameters, we can list:

- the output bitrate of the resulted video stream,
- the target bitrate of the Transport Stream,
- the target geometry of the stream (frame size),
- insert or not null packets into the transport stream (in order to make the bitrate constant),
- burn or not subtitles (DVB or Teletext) into the picture, their PID and Teletext page.

As advanced parameters we can list the usage of custom quantization matrices, different parameters for subsections of the transcoder, and logo insertion into the picture (frames).

The Transcoding servers are presented in details in [114, 113, 85].

## 6.2.4 Requirements and motivation

The main purpose of the management system is to allow the configuration, control and monitoring of heterogeneous systems, each system supporting several types of servers and many servers for each type of server.

The design of the system was based on a clear set of requirements. Some of these requirements have influenced the design process, others have influenced the implementation process: selecting the implementation environment, implementation language, etc. The main difference between the two sets of requirements is that the requirements influencing the implementation process have absolutely no influence in the design process. Below is a summary of the initial set of requirements. We also specify the set each requirement belongs to (design or implementation):

1. The system should be secure, the administrator access should require authentication and the communication between the client (browser) and the server should be encrypted (implementation).
2. The system should support several administrators, each having different capabilities and tasks (implementation)
3. Adding a server into the system should be simple and should not require any customization on the server side (such as the IP of the management server), (design)
4. Avoid Java wherever possible (implementation)
5. High reliability:
  - (a) Easy backup of configuration data (database backup) (implementation)
  - (b) The management server should be easily duplicable (implementation)
  - (c) If the management server fails, the servers in the system should be able to function normally (design)
  - (d) Reliability for servers in the system: standby servers. If a server (e.g. transcoder) has a hardware failure, then a standby server should take its place (design).
6. Modularity: if a type of servers is not present in a certain instance of the system, then all the resources associated with that type of servers should not be present (e.g. specific tables in the database, specific code, etc), (design).

Requirement 1 allows accessing the management server from anywhere in the world. Requirement 2 allows having several administrators, very useful in a big system where the administration work is divided between several persons. Requirement 3 allows a server to be added to a system after a generic installation of the software, without any configuration required on that server. This way, all the configuration is performed from the management server. Requirement 4 allows an administrator to use any browser and any operating system, since Java is not yet available on all platforms and all devices with a web browser, and there are compatibility problems between different browsers handling Java applications and also between Java implementations on different platforms. However, for one task (uploading a movie on a VoD server) Java could not be avoided and this task uses a Java applet. Requirement 5.a allows the backup of the data in the management server database, in case the server becomes unavailable. Requirement 5.b allows a management server to have a second instance (the server application also runs on a second machine, possibly using the same database as the main server), in case the main server becomes unavailable. Requirement 5.c allows each server to run independent of the management server. For this, each server has a local copy of the configuration data it needs and also accepts commands locally. Status information (log) is also recorded locally. Requirement 5.d allows a task to be done with increased reliability. For example, if we have a transcoder receiving data (an MPEG-2 stream) from a multicast IP address, transcoding this stream and sending the resulting stream to another multicast address, and if this transcoder fails (e.g. hardware failure), another standby server (transcoder) should take its place. If the initial server becomes available again, it becomes a standby server. This feature however is not possible for all types of servers. For example, if one IP TV server fails, a standby server may not have access to the same satellites as the failing server. The design of the requirement 5.d makes each server to multicast its status on a common, known address. Standby servers are listening to this address and when detecting that a server suddenly failed (is silent), one of them takes its place. Requirement 6 allows the management system to be used for managing only one or two types of servers, without consuming resources for the other, unused, server types and without presenting the administrators with unnecessary menus and configuration pages.

### 6.3 System architecture

The general architecture of the management platform is presented in Fig. 6.2.

The *Administrator* uses a web browser to communicate with the management platform. The web pages viewed by the *Administrator* come from the *Management Server*, which is a physical computer that hosts several components of the management platform. These pages are generated at runtime by the *Server Application* and sent to the *Administrator* using a *Web Server*. The *Server Application* communicates with the *Database* and with the *Physical Servers* in the system. If the *Administrator* configures some components of the system, the configuration information is saved by the *Server Application* in the *Database*. If the *Administrator* requests monitoring information about the system, this information is fetched from the *Physical Servers*

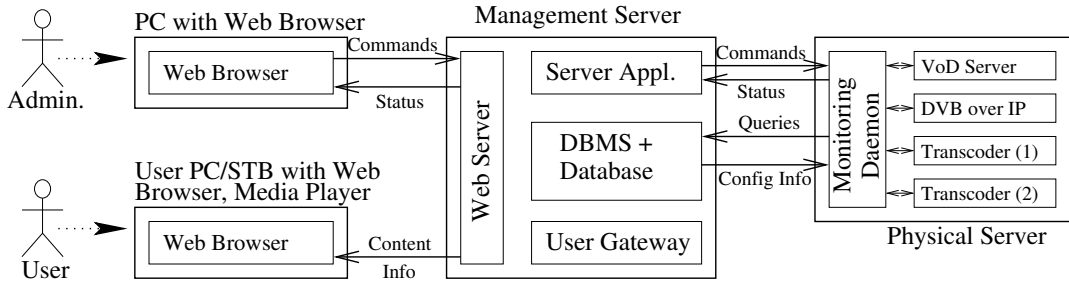


Figure 6.2: Components of the Management Platform.

and from the *Database* and it is presented in a web page to the *Administrator*. If the *Administrator* sends commands to servers in the system, the *Server Application* forwards the commands to the target *Physical Servers*.

A *Physical Server* is a PC that hosts a *Monitoring Daemon* and one or several multimedia servers. Multimedia server is a general term for an IP TV server, or VoD server or a Transcoding server. The *Monitoring Daemon* acts as an interface between the *Server Application* and the multimedia servers. It either executes the commands received from the *Server Application* (in the case or **Start Server** or **Stop Server**) or it forwards the received commands to their target server. The *Monitoring Daemon* is also responsible with monitoring the multimedia servers and sending the status information to the *Server Application* and *Database*.

The same *Management Server* provides content information for the users of the multimedia system. This content information is provided as web pages by the *Web Server*. The user web pages are generated by the *User Gateway* using status and content information stored in the *Database*.

### 6.3.1 The Server Application

As specific to all web applications, the *Server Application* generates HTML pages to be sent to the client browser. After generating such a page the execution of the application stops. The only communication between the current execution (the page that is currently generated) and previous executions (previous web pages) is performed through methods specific to HTTP: cookies and variables passing through GET and PUSH methods.

Each generated page has a certain structure and the execution process performs certain steps. First, the application registers all the available modules (Requirement 6) and a user menu is generated based on this information. Any web page presented to the user has one HTML form, containing some data that the user can change and send back (submit). Thus, depending on the last operation, the data presented to the user in this form comes from database, from the previous web page or from some *Monitoring Daemons*.

The information presented to the user through the generated web page has 3 purposes: configuration information, status information and possible commands. The data source for the configuration information is the database or previous web page.

After all the modifications have been made to the configuration information in a web page, the form is submitted and the information is saved in the database.

The status information is retrieved by the application directly from the *Monitoring Daemons*. The user can not modify this information, but the application may filter the displayed *Monitoring Daemon* commands out of all possible commands based on the status information. For example, depending if a server is stopped or running, the application will enable the start command or the stop and restart commands.

The commands issued by the user for the servers in the system are sent by the server application to corresponding *Monitoring Daemons*, using a text-based ad-hoc protocol. A *Monitoring Daemon* receives the command and it either executes it (if it has to start, stop or restart a server) or it forwards it to the corresponding server. The result of the command (and possibly an error message) are returned to the *Server Application*.

### 6.3.2 The Monitoring Daemon

The link between the *Server Application* and the servers running on one machine is made by the *Monitoring Daemon*. This is a light server running all the time on every server machine in the system. The necessity of having the *Monitoring Daemon* comes from three reasons:

1. To be able to start and stop a multimedia server from the management server
2. To simplify the access of the *Server Application* to the status information of multimedia servers running on one machine (the *Server Application* connects to a specific port on which the *Monitoring Daemon* is listening for connections).
3. To have all the common code related to the communication with the *Server Application* in one place, thus easier to modify and update.

The *Monitoring Daemons* are also managing the *Standby Servers*. They are the one listening for status information from other servers and starting a *Standby Server* when it needs to take the place of some server.

### 6.3.3 The Servers

Each server running on a machine in the system needs to communicate with the *Monitoring Daemon*. This communication is very simple, since normally, the server only sends periodically status information to its *Monitoring Daemon*.

However, certain servers have additional commands. One such example is the IP TV server, that has an additional command to detect the number and type of the DVB cards available to it, and to report this information through the *Monitoring Daemon* to the *Server Application*. Another example is the VoD server, that reports its available free space for new content and its current number of clients.

### 6.3.4 The User Gateway

The *User Gateway* is similar in design with the *Server Application*, however, it is different in functionality. Same as the *Server Application*, the *User Gateway* is a PHP application that generates HTML pages to be sent to the browsers of the users. The purpose of the *User Gateway* is to facilitate the browsing of the content available from the multimedia system. There are two main types of possible content:

- TV channels, streamed by IP TV servers or by Transcoding Servers,
- Movies/Content available in VoD mode from the VoD servers.

The *User Gateway* uses the same database as the *Server Application*. This database also stores information about users, including their viewing preferences. Using user preferences, the *Media Gateway* is able to present each user a personalized view of the available content. TV Channels can be ordered and sorted by their language and their genre (news, sports, music, etc.). Movies can be ordered and sorted by their language, genre (drama, adventure, Sci-Fi, etc.), rating (see Fig. 6.5).

## 6.4 Implementation details

The system was implemented using Linux, Apache web server, MySQL database (PostgreSQL also supported) and PHP for the *Server Application*. The *Monitoring Daemons* and the *Servers* are implemented in C/C++.

The Apache web server supports encrypted communication (using SSL) and it also supports authentication (fulfilling Requirement 1). The *Database Management System* (MySQL or PostgreSQL) also supports authentication. With the login and the password introduced by the user when connecting to the system, the *Server Application* tries to connect to the *Database*. If this connections succeeds, the user is authenticated. Thus, it is possible to have several administrators (users of the database) having different permissions to the tables in the database, thus limiting certain actions they can do in the system (Requirement 2).

Installing the system is easy, after required packages are installed (Apache, PHP, MySQL, etc.) a script creating the database and setting initial passwords has to be run. After this, the system is ready and the administration can connect and manage servers. The system also features a list of global preferences, which affect some default values for certain configuration items in the system and also the application's behavior in some places. The selected *Database Management System* (MySQL) is easy to backup (Requirement 5.a), just by making a backup of its data directory in the system (for Mandrake Linux this directory is /var/lib/mysql). The *Management Server* can be duplicated easily just by duplicating the original server's machine (Requirement 5.b). The backup *Management Server* may use the original server's database, or it can have its own database (periodically correlated with the original database).

The described system was designed by the author of this thesis and partially implemented by the author and his coworkers.



Figure 6.3: Screenshot of IP TV Server configuration: Assigning satellites to DVB-S cards.



Figure 6.4: Screenshot of IP TV Server configuration: Assigning Channels to DVB-S Cards.

## 6.5 Summary and author's contributions

In this chapter we addressed the problem of monitoring large-scale multimedia systems composed from heterogeneous devices from different providers. We also address the problem of client user interface for such a system, that typically is able to stream several types of content. Our solution to these problems is a web-based management platform able to configure, monitor and control several types of servers. The scientific contribution brought by this platform consists in its design, capable of managing several different servers, able of ensuring their reliability, and capable of presenting the

end-users with an attractive user interface to the content available from the system.

The work presented in this chapter is based on [82, 84, 85, 79, 78]. The author of this thesis contributed to this work by designing the management platform and by implementing it. The servers used by the presented system were implemented by several persons: the IP TV server by Aurelian Pop, the Transcoding server by Prakash Sastry, Mikko Suniala, Marius Vlad and Aurelian Pop, the VoD server by Marius Vlad.



Figure 6.5: Screenshot of the User Gateway: Available Movies.

# Chapter 7

## Conclusions

This thesis covered several issues related to IP-based VoD systems for streaming high-quality content. The motivation, the problems and the proposed solutions were introduced in Chapter 1. Chapters 2 to 6 proposed solutions to overcome some of the problems identified in Chapter 1.

In Chapter 2 we addressed the complexity issue of the media streaming part in VoD system components (clients or servers) when the streaming uses the RTSP, RTP and SDP protocols. The complexity of the streaming part comes from the internal communication between the RTSP, RTP and SDP protocols, which, in the absence of an integrated package, has to be handled by the application. The proposed solution consists of an integrated RTSP, RTP and SDP library that handles all the communication between the three protocols internally, providing a common interface to all these three protocols. The library also handles extensions to RTSP and SDP protocols, as allowed by their respective standards. Extending these protocols is done outside the library, at the application level, without the need to modify the library. The RTSP, RTP and SDP library has been implemented and tested experimentally in several applications. This library can be used to provide streaming services to any server or client of a multimedia system.

The used approach provides a solution to interoperability issues in VoD systems, by allowing a client to interoperate with several different servers, each with its own extensions and modifications to the control or transport protocol.

Future work regarding this library would include the integration of RTCP and studies for suitability in multicast environments.

In Chapter 3 we addressed the interoperability problem between components of a VoD system using DMIF and components using IETF protocols (RTSP, RTP and SDP). The incompatibility between the DMIF solution and the IETF-based solution comes from the fact that each solution is using different protocols for the transport and control of MPEG-4 streams. The solution proposed in this thesis maps the DMIF primitives onto the RTSP, RTP and SDP protocols, using them at the lower communication layers. The proposed solution also leverages the existing work and experience about IETF protocols by making RTSP, RTP and SDP an alternative to DMIF Default Signalling for communication between DMIF peers. The mapping was implemented as a DMIF plugin. This plugin can be used with any MPEG-4

application based on DMIF.

The approach used provides a solution to interoperability issues in VoD systems, by requiring minimal modifications (from networking point of view) from existing MPEG-2 clients and servers, in order to support MPEG-4 streams. We can envision situations when a client requests a stream from a server without initially knowing if it is an MPEG-2 or MPEG-4 stream. Our solution allows the client to do this. The client would find out the nature of the requested stream later, after performing several initialization steps.

Further work will include extensive tests in different VoD environments.

In Chapter 4 we addressed the problem of lack of players on the Linux platform for playing high-quality content from a VoD server. A video player able of playing MPEG-2 and able of network-based video retrieval was not existing when the player presented in this thesis was started. The player proposed here had to comply with several requirements, such as to be suitable for both STB and PC environments. Our adopted solution created a player with a modular design that fulfilled all the targeted requirements (as described Chapter 4) by having a modular architecture and by implementing all its media processing and control objects as plugins.

The used approach provided what it seemed to be the only solution for MPEG-2 playback over the network, on Linux systems, at the time it was ready. By using the RTSP library presented in Chapter 2, the player was able to interoperate with several VoD servers, some available commercially.

Future work regarding the player should focus on the integration with different Linux-based STB. Continuing the development for playing in PC-based Linux system is not recommended, due to the existence nowadays of better players for this environment.

In Chapter 5 we addressed the problem of lowering the price and the complexity of the content retrieval from the storage subsystem in a VoD server. We created a simple procedure for retrieving streams from the storage subsystem and a model capable of computing the retrieving capacity of the storage subsystem. The proposed approach demonstrates that it is possible to achieve the high performance required by a VoD server, when reading multimedia streams, with a PC running Linux and having an ATA RAID0. The maximum achieved performance was 500 Mbits/s, when using hardware RAID0, direct access method, and a block size of 1024 KB, and the number of disk seeks per second was 10. This speed is enough to serve more than 100 clients requesting MPEG-2 streams (at 4 Mbits/s). The tests showed that global system performance was highly correlated with the hardware and software configuration, and a correct tuning process was generally required in order to achieve the optimum performance.

The presented approach provides a solution for creating a VoD server, capable of streaming up to 40 streams, from an ordinary desktop PC. The same approach would require slightly better PC for streaming up to 100 streams. Another benefit of the approach is its simplicity: the implemented VoD server software works on any recent, standard, Linux installation. No special configuration is necessary, other than perhaps a RAID.

Future work will include finding a solution to use the proposed algorithm in

order to send fast-forward or fast-reverse streams, created in real time from standard streams. In addition to this, it is also possible to optimize further the proposed algorithm by ordering the streams scheduled for retrieval during one cycle. More future work could consist in extending the proposed algorithm to MPEG-4 streams.

In Chapter 6 we addressed the problem of system and servers management in a multimedia system. Such a multimedia system has several types of servers that provide several types of multimedia services. One such service is VoD. Managing such a system in an efficient way is important in order to provide a cost-effective way to configure, control and monitor the quality of offered services. Additionally, in Chapter 6 we addressed the problem of designing a suitable customer interface for such a system. This user interface has to be simple enough so that everybody can use the system, but it should also contain advanced features (such as customized search) for advanced users. Our approach to these problems was a web-based management platform which was able to configure, monitor and control several types of servers and was also capable of presenting the end-users with an attractive user interface to the content available from the system.

The used approach provides a simple and scalable solution for multimedia systems management. Our solution requires little support from the managed servers. It adds value to the managed servers and the system by providing additional features (such as the ease of configuration) that are independent on the basic servers existing in the system.

Future work will include reliability testing, adding new features such as automatic configuration of servers, automatic load balancing of VoD servers. In addition to this, it would be useful to tests the usability of the provided customer interface.

# Annex A

The complete log for RTSP messages exchanged between the MPEG-4 client and server presented in Chapter 3. The MPEG-4 content used is “Linda”. Timing information for each command is also present. This time represents the number of seconds passed since the initialization of the DMIF plugin until the corresponding command was processed by the RTSP, RTP and SDP library.

```
TCP Send(155) after      0.000 seconds :
ASCII: "DESCRIBE rtsp://localhost:5554/linda.mp4 RTSP/1.0
CSeq: 1
Accept: application/sdp
User-Agent: TTKK/DMI RTSP Client 1.1 (Linux/Win32)
DMIF-Session:9782400
```

"

```
TCP Recv(806) after      0.110 seconds :
ASCII: "RTSP/1.0 200 OK
CSeq: 1
Content-Length: 663
Content-Type: application/sdp
Server: TTKK/DMI RTSP Server 1.1 (Linux/Win32)
DMIF-Session:9782400
```

```
a=control:rtsp://localhost/linda.mp4
a=X-MPEG4-IOD:AoCAgHIAHwEBAgEBA4CAgCsAATAEgICADf8FAADIAAAAAAAAAAAGg
ICAEQakAAAD6AAAAGQgAAAAABd/A4CAgDYAAjAEgICAGP8NAAPoAAAAAAAAAAAFgICA
BlK4CwAJAAaAgIARACQAAABkAAAAZCAAAAAAF38=
m=video 0 RTP/AVP 96
a=rtpmap:96 mpeg4-sl/1000
a=control:rtsp://localhost/linda.mp4/streamid=1
a=mpeg4-esid:1
m=video 0 RTP/AVP 97
a=rtpmap:97 mpeg4-sl/100
a=control:rtsp://localhost/linda.mp4/streamid=2
a=mpeg4-esid:2
```

```
m=video 0 RTP/AVP 98
a=rtpmap:98 mpeg4-sl/1000
a=control:rtsp://localhost/linda.mp4/streamid=2115
a=mpeg4-esid:2115
m=audio 0 RTP/AVP 99
a=rtpmap:99 mpeg4-sl/1000
a=control:rtsp://localhost/linda.mp4/streamid=2113
a=mpeg4-esid:2113
"
UDP socket created
TCP Send(200) after      0.120 seconds :
ASCII: "SETUP rtsp://localhost:5554/linda.mp4/streamid=1 RTSP/1.0
CSeq: 2
Transport: RTP/AVP/UDP; unicast; client_port=1181; mode=play
User-Agent: TTKK/DMI RTSP Client 1.1 (Linux/Win32)
DMIF-Session:9782400

"
TCP Recv(194) after      0.120 seconds :
ASCII: "RTSP/1.0 200 OK
CSeq: 2
Session: 166072449660
Server: TTKK/DMI RTSP Server 1.1 (Linux/Win32)
Transport: RTP/AVP/UDP; unicast; server_port=1182; client_port=1181
DMIF-Session:9782400

"
UDP socket created
TCP Send(222) after      0.120 seconds :
ASCII: "SETUP rtsp://localhost:5554/linda.mp4/streamid=2 RTSP/1.0
CSeq: 3
Session: 166072449660
Transport: RTP/AVP/UDP; unicast; client_port=1183; mode=play
User-Agent: TTKK/DMI RTSP Client 1.1 (Linux/Win32)
DMIF-Session:9782400

"
TCP Recv(194) after      0.120 seconds :
ASCII: "RTSP/1.0 200 OK
CSeq: 3
Session: 166072449660
Server: TTKK/DMI RTSP Server 1.1 (Linux/Win32)
Transport: RTP/AVP/UDP; unicast; server_port=1184; client_port=1183
DMIF-Session:9782400
```

```
"
TCP Send(172) after      0.130 seconds :
ASCII: "PLAY rtsp://localhost:5554/linda.mp4/streamid=1 RTSP/1.0
CSeq: 4
Session: 166072449660
Scale: 1.00
User-Agent: TTKK/DMI RTSP Client 1.1 (Linux/Win32)
DMIF-Session:9782400

"
TCP Recv(115) after      0.130 seconds :
ASCII: "RTSP/1.0 200 OK
CSeq: 4
Session: 166072449660
Server: TTKK/DMI RTSP Server 1.1 (Linux/Win32)
DMIF-Session:9782400

"
TCP Send(172) after      0.130 seconds :
ASCII: "PLAY rtsp://localhost:5554/linda.mp4/streamid=2 RTSP/1.0
CSeq: 5
Session: 166072449660
Scale: 1.00
User-Agent: TTKK/DMI RTSP Client 1.1 (Linux/Win32)
DMIF-Session:9782400

"
TCP Recv(115) after      0.130 seconds :
ASCII: "RTSP/1.0 200 OK
CSeq: 5
Session: 166072449660
Server: TTKK/DMI RTSP Server 1.1 (Linux/Win32)
DMIF-Session:9782400

"
UDP socket created
TCP Send(225) after      0.160 seconds :
ASCII: "SETUP rtsp://localhost:5554/linda.mp4/streamid=2115 RTSP/1.0
CSeq: 6
Session: 166072449660
Transport: RTP/AVP/UDP; unicast; client_port=1185; mode=play
User-Agent: TTKK/DMI RTSP Client 1.1 (Linux/Win32)
DMIF-Session:9782400

"
```

TCP Recv(194) after 0.160 seconds :  
ASCII: "RTSP/1.0 200 OK  
CSeq: 6  
Session: 166072449660  
Server: TTKK/DMI RTSP Server 1.1 (Linux/Win32)  
Transport: RTP/AVP/UDP; unicast; server\_port=1186; client\_port=1185  
DMIF-Session:9782400

"

UDP socket created  
TCP Send(225) after 0.160 seconds :  
ASCII: "SETUP rtsp://localhost:5554/linda.mp4/streamid=2113 RTSP/1.0  
CSeq: 7  
Session: 166072449660  
Transport: RTP/AVP/UDP; unicast; client\_port=1187; mode=play  
User-Agent: TTKK/DMI RTSP Client 1.1 (Linux/Win32)  
DMIF-Session:9782400

"

TCP Recv(194) after 0.160 seconds :  
ASCII: "RTSP/1.0 200 OK  
CSeq: 7  
Session: 166072449660  
Server: TTKK/DMI RTSP Server 1.1 (Linux/Win32)  
Transport: RTP/AVP/UDP; unicast; server\_port=1188; client\_port=1187  
DMIF-Session:9782400

"

TCP Send(175) after 0.170 seconds :  
ASCII: "PLAY rtsp://localhost:5554/linda.mp4/streamid=2115 RTSP/1.0  
CSeq: 8  
Session: 166072449660  
Scale: 1.00  
User-Agent: TTKK/DMI RTSP Client 1.1 (Linux/Win32)  
DMIF-Session:9782400

"

TCP Recv(115) after 0.170 seconds :  
ASCII: "RTSP/1.0 200 OK  
CSeq: 8  
Session: 166072449660  
Server: TTKK/DMI RTSP Server 1.1 (Linux/Win32)  
DMIF-Session:9782400

"

TCP Send(175) after 0.170 seconds :  
ASCII: "PLAY rtsp://localhost:5554/linda.mp4/streamid=2113 RTSP/1.0  
CSeq: 9  
Session: 166072449660  
Scale: 1.00  
User-Agent: TTKK/DMI RTSP Client 1.1 (Linux/Win32)  
DMIF-Session:9782400

"

TCP Recv(115) after 0.170 seconds :  
ASCII: "RTSP/1.0 200 OK  
CSeq: 9  
Session: 166072449660  
Server: TTKK/DMI RTSP Server 1.1 (Linux/Win32)  
DMIF-Session:9782400

"

TCP Send(188) after 294.543 seconds :  
ASCII: "TEARDOWN rtsp://localhost:5554/linda.mp4/streamid=1 RTSP/1.0  
CSeq: 10  
Session: 166072449660  
User-Agent: TTKK/DMI RTSP Client 1.1 (Linux/Win32)  
DMIF-Session:9782400  
DMIF-Reason: REASON\_OK

"

TCP Recv(116) after 294.543 seconds :  
ASCII: "RTSP/1.0 200 OK  
CSeq: 10  
Session: 166072449660  
Server: TTKK/DMI RTSP Server 1.1 (Linux/Win32)  
DMIF-Session:9782400

"

TCP Send(188) after 294.543 seconds :  
ASCII: "TEARDOWN rtsp://localhost:5554/linda.mp4/streamid=2 RTSP/1.0  
CSeq: 11  
Session: 166072449660  
User-Agent: TTKK/DMI RTSP Client 1.1 (Linux/Win32)  
DMIF-Session:9782400  
DMIF-Reason: REASON\_OK

"

TCP Recv(116) after 294.543 seconds :  
ASCII: "RTSP/1.0 200 OK

CSeq: 11  
Session: 166072449660  
Server: TTKK/DMI RTSP Server 1.1 (Linux/Win32)  
DMIF-Session:9782400

"

TCP Send(191) after 294.553 seconds :  
ASCII: "TEARDOWN rtsp://localhost:5554/linda.mp4/streamid=2113 RTSP/1.0  
CSeq: 12  
Session: 166072449660  
User-Agent: TTKK/DMI RTSP Client 1.1 (Linux/Win32)  
DMIF-Session:9782400  
DMIF-Reason: REASON\_OK

"

TCP Recv(116) after 294.553 seconds :  
ASCII: "RTSP/1.0 200 OK  
CSeq: 12  
Session: 166072449660  
Server: TTKK/DMI RTSP Server 1.1 (Linux/Win32)  
DMIF-Session:9782400

"

TCP Send(191) after 294.553 seconds :  
ASCII: "TEARDOWN rtsp://localhost:5554/linda.mp4/streamid=2115 RTSP/1.0  
CSeq: 13  
Session: 166072449660  
User-Agent: TTKK/DMI RTSP Client 1.1 (Linux/Win32)  
DMIF-Session:9782400  
DMIF-Reason: REASON\_OK

"

TCP Recv(116) after 294.553 seconds :  
ASCII: "RTSP/1.0 200 OK  
CSeq: 13  
Session: 166072449660  
Server: TTKK/DMI RTSP Server 1.1 (Linux/Win32)  
DMIF-Session:9782400

"

TCP Send(177) after 294.563 seconds :  
ASCII: "TEARDOWN rtsp://localhost:5554/linda.mp4 RTSP/1.0  
CSeq: 14  
Session: 166072449660  
User-Agent: TTKK/DMI RTSP Client 1.1 (Linux/Win32)

DMIF-Session:9782400  
DMIF-Reason: REASON\_OK

"

TCP Recv(116) after 294.563 seconds :  
ASCII: "RTSP/1.0 200 OK  
CSeq: 14  
Session: 166072449660  
Server: TTKK/DMI RTSP Server 1.1 (Linux/Win32)  
DMIF-Session:9782400

"

TCP Recv(0) after 294.563 seconds :  
ASCII: ""  
We lost the session socket !

# Bibliography

- [1] C.C. Aggarwal, J.L. Wolf, and P.S. Yu. A permutation-based pyramid broadcasting scheme for video-on-demand systems. In *Proc. of IEEE International Conference on Multimedia Computing and Systems*, pages 118–126, June 1996.
- [2] C.C. Aggrawal, J.L. Wolf, and P.S. Yu. On Optimal Piggyback Merging Policies for Video-on-Demand Systems. In *Proc. of ACM Sigmetrics '96*, Philadelphia, USA, May 1996.
- [3] T. Ahmed, G. Buridant, and A. Mehaoua. Delivering of MPEG-4 Multimedia Content over Next Generation Internet. In *Proc. of 4th IFIP/IEEE International Conference on Management of Multimedia Networks and Services (MMNS)*, Chicago, IL, USA, Oct–Nov 2001.
- [4] T. Ahmed, G. Buridant, and A. Mehaoua. Encapsulation and marking of MPEG-4 video over IP differentiated services. In *Proc. of Sixth IEEE Symposium on Computers and Communications*, pages 346–352, 2001.
- [5] T. Ahmed, A. Mehaoua, and R. Boutaba. Interworking between SIP and MPEG-4 DMIF for heterogeneous IP video. In *Proc. of IEEE International Conference on Communications (ICC)*, volume 4, pages 2469–2473, 2002.
- [6] T. Ahmed, A. Mehaoua, and G. Buridant. Implementing MPEG-4 video on demand over IP Differentiated Services. In *Proc. of Global Telecommunications Conference*, volume 4, pages 2489–2493, 2001.
- [7] C. Aurrecoecchea, A.T. Campbell, and L. Hauw. A survey of QoS architectures. *Multimedia Systems*, 6:138–151, 1998.
- [8] V. Balabanian. The Role of DMIF with RTSP and MPEG-4. Technical report, ISO/IEC/JTC1/SC29/WG11: MPEG98/M4102, October 1998; containing: draft-balabanian-rtsp-mpeg4-dmif-00.txt , Sep 22, 1998, 1998.
- [9] A. Basso and & al. Preliminary results in streaming MPEG-4 over IP with the MPEG-4/IETF-AVT payload format. Technical report, Technical report MPEG Systems Group, Melbourne, Australia, 1999.
- [10] A. Basso, R. Civanlar, P. Gentric, Herpel, Lifshitz, Lim, Perkins, and J. van Der Meer. RTP Payload Format for MPEG-4 Streams. Technical report, IETF Internet Draft draft-ietf-avt-mpeg4-multisl-04.txt, Feb 2002.

- [11] A. Basso and S. Varakliotis. Transport of MPEG-4 over IP/RTP. In *Proc. of International Conference on Multimedia and Expo (ICME)*, volume 3, pages 1067–1070, New York, NY, USA, Aug.
- [12] A. Basso, S. Varakliotis, R. Castagno, and F. Lohan. Transport of MPEG-4 over IP/RTP. *European Transactions on Telecommunications*, 12(3), May–Jun 2001.
- [13] T. Berners-Lee and & al. Uniform Resource Identifiers (URI): Generic Syntax. Technical report, Technical Report RFC 2396, Internet Engineering Task Force (IETF), Aug 1998.
- [14] C. Bertin, I. Defée, J. Kangasoja, and M. Rustari. Commerce with MPEG-4 on the Internet with QoS. In *Proc. of IEEE International Conference on Multimedia Computing and Systems*, volume 2, pages 1043–1045, Jul 1999.
- [15] A.P. Black, Jie Huang, R. Koster, J. Walpole, and Calton Pu. Infopipes: An abstraction for multimedia streaming. *Springer-Verlag*, 8:406–419, 2002.
- [16] A.T. Campbell and G. Coulson. A qos adaptive transport system: design, implementation and experience. In *Proc. of the fourth ACM international conference on Multimedia*, pages 117–127, 1997.
- [17] A.T. Campbell and S. Keshav. Guest editorial - quality of service in distributed systems. *Computer Communications Journal*, 21(4):291–293, Apr 1997.
- [18] S.W. Carter and D.D.E. Long. Improving video-on-demand server efficiency through stream tapping. In *Proc. of International Conference on Computer Communications and Networks*, pages 200–207, September 1997.
- [19] J. Case, M. Fedor, M. Schoffstal, and J. Davin. A Simple Network Management Protocol (SNMP). Technical report, Technical Report RFC 1157, Internet Engineering Task Force (IETF), Network Working Group, May 1990.
- [20] R. Castagno, S. Kiranyaz, F. Lohan, and I. Defée. An Architecture Based on IETF Protocols for the Transport of MPEG-4 Content over the Internet. In *Proc. of ICME*, volume 3, pages 1322–1325, New York, NY, USA, Aug 2000.
- [21] R. Castagno, F. Lohan, and G. Franceschini. Text for ISO/IEC 14496-1 Version 3 VM 9.0. Technical report, MPEG-4 verification model, document no. N3198p4, 2000.
- [22] Jun Chen, Jianxin Liao, and Junliang Chen. A novel active multi-layer fuzzy decision algorithm based on multi-factor for VoD. In *Proc. of Fifth Asia-Pacific Conference on Comm.*, volume 2, pages 1010–1014, Oct 1999.
- [23] T. Chiueh and C. Lu. A periodic broadcasting approach to video-on-demand service. *International Society for Optical Engineering*, 2615:162–169, 1995.

- [24] P. Christ, C. Guillemot, and S. Wesner. RTSP-based Stream Control in MPEG-4. Technical report, IETF Internet Draft draft-christ-rtsp-mpeg4-00.txt, Nov 1998.
- [25] A. Dan, D. Sitaram, and P. Shahabuddin. Scheduling policies for an on-demand video server with batching. In *Proc. of ACM Multimedia*, pages 15–23, San Francisco, CA, Oct 1994.
- [26] A. Dan, D. Sitaram, and P. Shahabuddin. Dynamic batching policies for an on-demand video server. *Multimedia Systems, Springer-Verlag*, 4:112–121, 1996.
- [27] D.L. Eager, M.K. Vernon, and J. Zahorjan. Optimal and efficient merging schedules for video-on-demand servers. In *Proc. of ACM Multimedia '99*, Orlando, FL, November 1999.
- [28] D.L. Eager, M.K. Vernon, and J. Zahorjan. Minimizing bandwidth requirements for on-demand data delivery. *IEEE Transactions on Knowledge and Data Engineering*, 2001.
- [29] R. Fielding and & al. Hypertext Transfer Protocol – HTTP/1.1. Technical report, Technical Report RFC 2616, Internet Engineering Task Force (IETF), Jun 1999.
- [30] G. Franceschini. General DMIF Software Architecture. Technical report, JUPITER II (Joint Usability, Performability and Interoperability Trials in Europe) Report, <http://www.eurescom.de/public-webospace/P800-series/P807/results/DMIF/R1/D2-T6-DMIF-R1-SoftwareArchitecture.pdf> (active in Nov 2003), Jan 1999.
- [31] L. Gao, J. Kurose, and D. Towsley. Efficient schemes for broadcasting popular videos. In *Proc. of International Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 317–329, August 1998.
- [32] L. Gao and D. Towsley. Supplying instantaneous video-on-demand services using controlled multicast. *IEEE Multimedia*, pages 117–121, 1999.
- [33] S. Ghandeharizadeh, S. Kim, and C. Shahabi. On configuring a single disk continuous media server. In *Proc. of ACM SIGMETRICS Conference*, May 1995.
- [34] S. Ghandeharizadeh, R. Zimmermann, W. Shi, R. Rejaie, D. Ierardi, and T.Li. Mitra. A scalable continuous media server. Technical report, Technical Report USC-CS-TR96-628, University of Southern California, Los Angeles, CA, August 1996.
- [35] G. Goldszmidt. *Distributed Management by Delegation*. PhD thesis, Columbia University, New York, NY, USA, Dec 1995.

- [36] L. Golubchik, J.C.S. Lui, and R. Muntz. Reducing I/O Demand in Video-on-Demand Storage Servers. In *Proc. of ACM Sigmetrics '95*, Ottawa, Canada, May 1995.
- [37] IETF Audio/Video Transport Group. <http://www.ietf.org/html.charters/avt-charter.html> (active in Nov 2003).
- [38] IETF Multiparty Multimedia Session Control Group. <http://www.ietf.org/html.charters/mmusic-charter.html> (active in Nov 2003).
- [39] Moving Pictures Experts Group. Coding Of Audio-Visual Objects (MPEG-4): MPEG-4 Specific Format - MP4. ISO/IEC 14496-1.
- [40] Moving Pictures Experts Group. Generic Coding Of Moving Pictures and Associated Audio (MPEG-2). ISO/IEC 13818, 1994.
- [41] Moving Pictures Experts Group. Coding Of Audio-Visual Objects: Reference Software (IM1-2D). ISO/IEC 14496-5 Draft Text of Final Draft International Standard, ISO/IEC JTC1/SC29/WG11 N2505, Nov 1998.
- [42] Moving Pictures Experts Group. Coding Of Audio-Visual Objects (MPEG-4). ISO/IEC 14496, 1999.
- [43] Moving Pictures Experts Group. Coding Of Audio-Visual Objects (MPEG-4): Delivery Multimedia Integration Framework. ISO/IEC 14496-6, 1999.
- [44] Object Management Group. The common object request broker: Architecture and specification. Technical report, Jul 1995.
- [45] C. Guillemot, P.Christ, and S. Wesner. RTP payload format for MPEG-4 Visual Advanced Profiles. Technical report, IETF Internet Draft draft-gc-avt-mpeg4visual-00.txt, Mar 2000.
- [46] C. Guillemot, P.Christ, S. Wesner, and A. Klemets. RTP Payload Format for MPEG-4 with Flexible Error Resilience. Technical report, IETF Internet Draft draft-ietf-avt-mpeg4streams-00.txt, 2000.
- [47] K.A. Haghghi, Y. Pourmohammadi, and H.M. Alnuweiri. Realizing MPEG-4 streaming over the Internet: a client/server architecture using DMIF. In *Proc. of International Conference on Information Technology: Coding and Computing*, pages 23–29, Apr 2001.
- [48] K.A. Haghghi, Y. Pourmohammadi, and H.M. Alnuweiri. Delivery of MPEF-4 object based multimedia in a multicast environment. In *Proc. of International Conference on Information Technology: Coding and Computing*, pages 446–451, 2002.

- [49] M. Handley and V. Jacobson. Session Description Protocol (SDP). Technical report, Technical Report RFC 2327, Internet Engineering Task Force (IETF), MMUSIC Working Group, April 1998.
- [50] M. Handley, C. Perkins, and E. Whelan. Session Announcement Protocol (SAP). Technical report, Technical Report RFC 2974, Internet Engineering Task Force (IETF), MMUSIC Working Group, October 2000.
- [51] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg. Session Initiation Protocol (SIP). Technical report, Technical Report RFC 2543, Internet Engineering Task Force (IETF), MMUSIC Working Group, March 1999.
- [52] B.G. Haskell, A. Puri, and A.N. Netravali. *Digital Video: An Introduction to MPEG-2*. Chapman and & Hall, 1997.
- [53] R. Haskin and F. Schmuck. The tiger shark file system. In *Proc. of IEEE CompCon*, pages 226–231, Santa Clara, CA, February 1996.
- [54] D. Hoffman and & al. RTP Payload Format for MPEG1/MPEG2 Video. Technical report, Technical Report RFC 2250, Internet Engineering Task Force (IETF), Audio/Video Transport, Jan 1998.
- [55] Yu Hongtao, Chor Ping Low, and Y. Atif. Design issues on video-on-demand resource management. In *Proc. of IEEE Int. Conf. on Networks*, pages 199–203, Sep 2000.
- [56] K. Hua, Y. Cai, and S. Sheu. Patching: A multicast technique for true video-on-demand services. *ACM Multimedia*, pages 191–200, 1998.
- [57] K.A. Hua, Y. Cai, and S. Sheu. Skyscraper broadcasting: A new broadcasting scheme for metropolitan video-on-demand systems. In *Proc. of ACM SIGCOMM*, pages 118–126, September 1997.
- [58] J.F. Huard, A.A. Lazar, Koon-Seng Lim, and G.S. Tselikis. Realizing the MPEG-4 multimedia delivery framework. *IEEE Network*, 12(6):35–45, Nov–Dec 1998.
- [59] ISMA. Internet Streaming Media Alliance. <http://www.isma.tv> (active in February 2004).
- [60] F.C. Jeng, M. Jeanson, S.Y. Zhu, and K. Konstantinides. Design of a home media center with network and streaming capabilities. In *Proc. of IEEE International Conference on Consumer Electronics (ICCE)*, pages 102–103, Jun 2002.
- [61] S. Josefsson. The base16, base32, and base64 data encodings. Technical report, Technical Report RFC 3548, Internet Engineering Task Force (IETF), July 2003.

- [62] Li-Shen Juhn and Li-Ming Tseng. Fast broadcasting for hot video access. In *Proc. - Fourth International Workshop on Real-Time Computing Systems and Applications*, pages 237–243, October 1997.
- [63] Li-Shen Juhn and Li-Ming Tseng. Fast data broadcasting and receiving scheme for popular video service. *IEEE Transactions on Broadcasting*, 44:100–105, March 1998.
- [64] H. Kalva, Li Tang, J.F. Huard, G. Tselikis, J. Zamora, Lai-Tee Cheok, and A. Eleftheriadis. Implementing multiplexing, streaming, and server interaction for MPEG-4. *IEEE Transactions on Circuits and Systems for Video Technology*, 9(8):1299–1311, 1999.
- [65] J. Kangasoja, M. Rustari, F. Lohan, and I. Defée. Media Server for Interactive MPEG-4 Applications on the Internet. In *Proc. of FINSIG*, pages 209–213, Oulu, Finland, 1999.
- [66] Y. Kikuchi, T. Nomura, S. Fukunaga, Y. Matsui, and H. Kimata. RTP Payload Format for MPEG-4 Audio/Visual Streams. Technical report, Technical Report RFC 3016, Internet Engineering Task Force (IETF), Audio/Video Transport, Nov 2000.
- [67] H. Kulander. History of Video in Linux. Technical report, <http://folk.uio.no/infmkt/mkt10b-lin.pdf> (active in Nov 2003).
- [68] D. Lee, N. Kim, and S. Kim. The MPEG-4 streaming player using adaptive Decoding Time Stamp. In *Proc. of Ninth International Conference on Parallel and Distributed Systems*, pages 398–403, Dec 2002.
- [69] J.Y.B. Lee. On a unified architecture for video-on-demand services. *IEEE Trans. on Multimedia*, 4(1):38–47, Mar 2002.
- [70] J.Y.B. Lee and C.H. Lee. Design, performance analysis, and implementation of a super-scalar video-on-demand system. *IEEE Trans. on Circuits and Systems for Video Technology*, 12(11):983–987, Nov 2002.
- [71] P.C.H Lee. Performance analysis of an MPEG-II audio/video player. *IEEE Transactions on Consumer Electronics*, 45(1):141–150, Feb 1999.
- [72] S. Leffler, M. McKusick, M. Karels, and J. Quarterman. *The design and Implementation of the 4.3 BSD Unix Operating System*. Addison-Wesley, 1989.
- [73] M. Linetsky. *Programming Microsoft DirectShow*. Wordware Publishing, 2002.
- [74] Viodi LLC. Video on demand - service consideration. [http://www.tripleplay.tv/pdf/Viodi-VOD\\_Service\\_Considerations.pdf](http://www.tripleplay.tv/pdf/Viodi-VOD_Service_Considerations.pdf) (active in Jan 2004), Jul 2001.
- [75] F. Lohan and I. Defée. Performance of High-Speed Media Server - ATM Network Interface. In *Proc. of FINSIG*, pages 276–280, Oulu, Finland, 1999.

- [76] F. Lohan and I. Defée. Modularity in open media terminal system architecture. In *Proc. of IEEE International Conference on Multimedia and Expo*, pages 708–711, Tokyo, Japan, Aug 2001.
- [77] F. Lohan and I. Defée. Open media terminal system. In *Proc. of International Conference on Media Futures*, pages 153–156, Florence, Italy, May 2001.
- [78] F. Lohan and I. Defée. Integrated web-based management system for a heterogeneous multimedia system. *Recent Advances in Communications and Computer Science*, WSEAS Press, pages 392–397, 2003.
- [79] F. Lohan and I. Defée. Integrated web-based management system for a heterogeneous multimedia system. In *Proc. of the 7th WSEAS International Multiconference on Circuits, Systems, Communications and Computers*, Corfu Island, Greece, July 2003.
- [80] F. Lohan, I. Defée, R. Castagno, and S. Kiranyaz. Content Delivery and Management in Networked MPEG-4 System. In *Proc. of EUSIPCO*, volume 4, pages 2313–2316, Tampere, Finland, Sep 2000.
- [81] F. Lohan, I. Defée, and H. Hakulinen. Design and Implementation of an Open Broadband Multimedia System. In *Proc. of Packet Video Workshop*, pages 246–255, Kyongju, Korea, Apr–May 2001.
- [82] F. Lohan, I. Defée, and H. Hakulinen. Networked Multimedia System Based on Open Architecture. In *Proc. of International Conference on Consumer Electronics*, pages 344–345, Los Angeles, CA, USA, Jun 2001.
- [83] F. Lohan, I. Defée, and M. Vlad. The Architecture of an Integrated RTSP, RTP and SDP Library. In *Proc. of IEEE International Conference on Telecommunications*, pages 338–342, Bucharest, Romania, Jun 2001.
- [84] F. Lohan, I. Defée, M. Vlad, and A. Pop. Integrated System for Multimedia Delivery Over Broadband IP Networks. In *Proc. of IEEE International Conference on Consumer Electronics (ICCE)*, pages 166–167, Los Angeles, CA, USA, Jun 2002.
- [85] F. Lohan, I. Defée, M. Vlad, A. Pop, and P. Sastry. Integrated System for Multimedia Delivery Over Broadband IP Networks. *IEEE Transactions on Consumer Electronics*, 48(3):564–574, Aug 2002.
- [86] F. Lohan, P. Sastry, and I. Defée. Broadband Network Set-Top Box System. In *Proc. of Protocols for Multimedia Systems (PROMS) Conference*, pages 257–263, Cracow, Poland, Oct 2000.
- [87] F. Lohan, M. Vlad, and I. Defée. Analysis and optimization of disk retrieval techniques in a PC-based VoD server. In *Proc. of the 7th WSEAS International Multiconference on Circuits, Systems, Communications and Computers*, Corfu Island, Greece, July 2003.

- [88] P. Lougher and D. Shepherd. The design of a storage server for continuous media. *The Computer Journal*, 36(1):32–42, 1993.
- [89] A. Mahanti. On-demand media streaming on the internet: Trends and issues. [citeseer.nj.nec.com/565045.html](http://citeseer.nj.nec.com/565045.html).
- [90] M. Meechan. Automating the digital broadcast process: control, complexity and cost. *IEEE Electronics & Communication Engineering Journal*, 13(3):101–116, Jun 2001.
- [91] Microsoft. DirectX. <http://www.microsoft.com/directx> (active in Nov 2003).
- [92] Microsoft. Introduction to DirectShow. <http://www.microsoft.com/Developer/PRODINFO/directx/dxm/help/ds/default.htm> (active in Nov 2003).
- [93] Sun Microsystems. Java Media Framework (JMF). <http://java.sun.com/products/java-media/jmf> (active in Nov 2003).
- [94] A.M. Mohamed and H.M. Alnuweiri. MPEG-4 broadcast: a client/server framework for multi-service streaming using push channels. In *Proc. of IEEE Fourth Workshop on Multimedia Signal Processing*, pages 543–548, 2001.
- [95] A. Nagasaka. Multimedia streaming by broadband connection - future prospects and development strategy at Oki Electric. *OKI Technical Review Journal*, 69(4):6–9, Oct 2002.
- [96] DivX Networks. Divx codec. <http://www.divx.com> (active in Nov 2003).
- [97] M. Ohlenroth and H. Hellwagner. RTP packetization of MPEG-4 elementary streams. In *Proc. of International Conference on Multimedia and Expo (ICME)*, volume 2, pages 465–468, 2002.
- [98] J.F. Paris. A simple low-bandwidth broadcasting protocol for video-on-demand. In *Proc. of Eight International Conference on Computer Communications and Networks*, pages 118–123, October 1999.
- [99] J.F. Paris, S.W. Carter, and D.D. Long. A hybrid broadcasting protocol for video on demand. In *Proc. of Multimedia Computing and Networking Conference*, pages 317–326, 1999.
- [100] M. Podesta, S. Giordano, and P. Cremonese. An Example of Traffic-Accommodating Application. In *Proc. of International Conference on Software, Telecommunications and Computer Networks*, Oct 2000.
- [101] A. Pop, F. Lohan, and I. Defée. Multicast Server for Distribution of Digital TV Streams over IP Networks. In *Proc. of URSI-FWCW*, pages 188–189, Tampere, Finland, Oct 2001.

- [102] Y. Pourmohammadi, K.A. Haghghi, A. Mohamed, and H.M. Alnuweiri. Streaming MPEG-4 over IP and Broadcast Networks: DMIF Based Architectures. In *Proc. of The 11th International Packet Video Workshop*, pages 218–227, Kyongju, Korea, Apr–May 2001.
- [103] GStreamer Open Source Project. GStreamer Multimedia Framework. <http://www.gstreamer.net> (active in Nov 2003).
- [104] MPlayer Open Source Project. MPlayer Linux Video Player.
- [105] Ogle Open Source Project. Ogle Linux DVD Player. <http://www.dtek.chalmers.se/groups/dvd> (active in Nov 2003).
- [106] XINE Open Source Project. XINE Linux Video Player. <http://xinehq.de> (active in Nov 2003).
- [107] XMMS Open Source Project. X MultiMedia System (XMMS). <http://www.xmms.org> (active in Nov 2003).
- [108] XMPS Open Source Project. X Movie Player System (XMPS).
- [109] Xvid Open Source Project. Xvid codec. <http://www.xvid.org> (active in Nov 2003).
- [110] K. Ramkishor and J.P. Mammen. Bandwidth Adaptation for MPEG-4 Video Streaming over the Internet. In *Proc. of Digital Image Computing Techniques and Applications (DICTA2002)*, Melbourne, Australia, Jan 2002.
- [111] A. Reddy and J. Wylie. Disk scheduling in a multimedia system. In *Proc. of ACM Multimedia Conference*, August 1993.
- [112] A. Reddy and J. Wylie. I/O issues in a multimedia system. *IEEE Computer*, 27(3):69–74, 1994.
- [113] P. Sastry and I. Defée. Real-time digital tv playback on handheld operating over w-lan. In *Proc. of International Conference on Third Generation Wireless and Beyond*, pages 635–639, San Francisco, CA, May 2002.
- [114] P. Sastry, A. Pop, M. Suniala, M. Vlad, and I. Defée. Software transcoding of digital tv streams. In *Proc. of the 2003 Finnish Signal Processing Symposium, FINSIG'03*, pages 144–148, Tampere, Finland, May 2003.
- [115] K. Sato, T. Takada, S. Aoyagi, T. Hirotsu, and T. Sugawara. Dynamic multimedia integration with the WWW. In *Proceedings of IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pages 448–451, Aug 1999.
- [116] H. Schulzrinne. RTP information about implementations. <http://www.cs.columbia.edu/hgs/rtp> (active in Nov 2003).

- [117] H. Schulzrinne. RTSP information about implementations. <http://www.cs.columbia.edu/hgs/rtsp/implementations.html> (active in Nov 2003).
- [118] H. Schulzrinne and & al. Real Time Streaming Protocol (RTSP). Technical report, Technical Report RFC 2326, Internet Engineering Task Force (IETF), MMUSIC Working Group, Apr 1998.
- [119] H. Schulzrinne and & al. RTP: A transport protocol for real-time applications. Technical report, Internet draft, Internet Engineering Task Force (IETF), Jul 14 2000.
- [120] H. Schulzrinne and & al. Real Time Streaming Protocol (RTSP) draft-ietf-mmusic-rfc2326bis-04.txt. Technical report, IETF Internet Draft, Jun 2003.
- [121] D. Singer and Y. Lim. A Framework for the delivery of MPEG-4 over IP-based Protocols. Technical report, IETF Internet Draft draft-singer-mpeg4-ip-04, Feb 1001.
- [122] D. Sitaram and A. Dan. *Multimedia Servers*. Morgan Kaufmann Publishers, 2000.
- [123] M. Smolenski, T. Fink, K. Konstantinides, D. Frankenberger, and C. Peplinski. Design of a personal digital video recorder/player. In *Proc. of IEEE Workshop on Signal Processing Systems (SiPS)*, pages 3–12, Oct 2000.
- [124] K. Soejima, M. Matsuda, T. Iino, T. Hayashi, and T. Nakajima. Building audio and visual home appliances on Linux. In *Proc. of IEEE Symposium on Applications and the Internet*, pages 25–30, Feb 2002.
- [125] Loki Software. Simple directmedia layer. <http://www.libsdl.org> (active in Nov 2003).
- [126] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1997.
- [127] T. Teory and T. Pinkerton. A comparative analysis of disk scheduling policies. *Communications of the ACM*, 15(3):177–184, March 1972.
- [128] Yi-Shin Tung and Ja-Ling Wu. Architecture design, system implementation, and applications of MPEG-4 systems. In *Proc. of Workshop and Exhibition on MPEG-4*, pages 37–40, Jun 2001.
- [129] Yi-Shin Tung, Ja-Ling Wu, and Ho Chia-Chiang. Architecture design of an MPEG-4 system. In *Proc. of IEEE International Conference on Consumer Electronics (ICCE)*, pages 122–123, Jun 2000.
- [130] J. van der Meer, D. Mackie, V. Swaminathan, D. Singer, and P. Gentric. RTP Payload Format for Transport of MPEG-4 Elementary Streams. Technical report, IETF Internet Draft draft-ietf-avt-mpeg4-simple-07.txt, Feb 2003.

- [131] E. Vayias, J. Soldatos, N. Mitrou, K. Kontovassilis, and G. Kormentzas. Managing networks over the web: Classification of approaches and an implementation. In *International Conference on Telecommunications*, pages 451–456, Chalkidiki, Greece, Jun 1998.
- [132] S. Viswanathan and T. Imielinski. Metropolitan area video-on-demand service using pyramid broadcasting. *Multimedia Systems, Springer-Verlag*, 4:197–208, 1996.
- [133] S. Wenger, M. Hannuksela, and T. Stockhammer. RTP payload Format for JVT Video. Technical report, IETF Internet Draft draft-ietf-avt-rtp-h264-01.txt, Feb 2003.
- [134] P. Westerink, L. Amini, S. Veliah, and W. Belknap. A live intranet distance learning system using MPEG-4 over RTP/RTSP. In *Proc. of International Conference on Multimedia and Expo (ICME)*, volume 2, pages 601–604, New York, NY, USA, Aug 2000.
- [135] Chun-Hsin Wu, Ann-Tzung Cheng, Shao-Ting Lee, and Jan-Ming Ho. An AutoPC for supporting in-vehicle navigation and location-based multimedia services. In *Proc. of IEEE Position Location and Navigation Symposium*, pages 226–232, Apr 2002.
- [136] H. Yamamoto. Multimedia streaming technology in broadband networks - 3. Digital right management system. *OKI Technical Review Journal*, 69(4):54–59, Oct 2002.
- [137] Ming-Hour Yang, Chi-He Chang, and Yu-Chee Tseng. A borrow-and-return model to reduce client waiting time for broadcasting-based vod services. *IEEE Transactions on Broadcasting*, 49(2):162–169, June 2003.
- [138] P. Yu, M. Chen, and D. Kandlur. Grouped sweeping scheduling for dasd-based multimedia storage management. *Multimedia Systems*, 1:99–109, 1993.
- [139] T. Zhang, S. Covaci, and R. Popescu-Zeletin. Intelligent agents in network and service management. In *Proc. of IEEE Global Telecommunications Conference (GLOBECOM)*, volume 3, pages 1855–1861, London, UK, Nov 1996.
- [140] R. Zimmermann, K. Fu, C. Shahabi, D. Yao, and H. Zhu. Yima: Design and evaluation of a streaming media system for residential broadband services. In *Lecture Notes in Computer Science*, volume 2209, pages 116–125, 2001.
- [141] M. Zink, C. Griwodz, and R. Steinmetz. Kom player-a platform for experimental VoD research. In *Proc. of IEEE Computers and Communications*, pages 370–375, Jul 2001.