Distribution Agnostic Video Server

Waqs Daar
Spring, 2010

Master of Science Thesis
Stockholm, Sweden 2010

TRITA-ICT-EX-2010:92
Distribution Agnostic Video Server

by

Waqas Daar

A thesis submitted in partial fulfillment for the degree of Master of Science in Internetworking

in the
Telecommunication System Laboratory (TSlab)
School of Information and Communication Technology

May 2010
Declaration of Authorship

I, Waqas Daar, declare that this thesis titled, ‘Distribution Agnostic Video Server’ and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:
KTH ROYAL INSTITUTE OF TECHNOLOGY

Abstract

Telecommunication System Laboratory (TSlab)
School of Information and Communication Technology

Master of Science in Internetworking

by Waqas Daar

With the advances of network and communication technology, real time audio and video streaming services are becoming progressively popular over the Internet. In order to enable universal access of multimedia streaming content and thus the desired end-to-end QoS, it is very desirable to design a video server. A video server, that can dynamically coupled to different streaming engines and deployed in a test bed for conducting different streaming experiments.

In this thesis we present the design of a video server that implement an "engine-agnostic" abstraction that will help to automate and repeat deterministic streaming experiments using different engines. Proposed video server is also deployed in a test bed for evaluating different performance measurement parameters like CPU load, memory utilization etc. The results of test bed also support our proposed idea and unfold many opportunities for the research community to perform different multimedia streaming experiments with proposed video server.
KTH ROYAL INSTITUTE OF TECHNOLOGY

Abstract

Telecommunication System Laboratory (TSlab)
School of Information and Communication Technology

Master of Science in Internetworking

by Waqas Daar


I denna avhandling presenterar vi designen av en video server att genomföra en ”motoragnostiker ”abstraktion som bidrar till att automatisera och upprepa deterministiska streaming försök med di erent motorer. Föreslagen video server är ocksåanserats i en provbänk för utvärdering di erent parametrar resultatmtning som CPU belastning, minne jande etc. Resultatet av en provbänk stöder ocks våra föreslagna id och fall många möjligheter för forskningen att genomföra di erent multimedia ström experiment med föreslagna video server.
Acknowledgements

This thesis work has been carried out in Department of Information Engineering and Computer Science in University of Trento, Italy.

First of all I would like to thank ALLAH Subhanahu wa-ta’ala for his help and then I would like to record my gratitude to both of my supervisors Renato Lo Cigno and Luca Abeni for their supervision, advice, and guidance from the very early stage of this research as well as giving me extraordinary experiences through out the work. Above all and the most needed, they provided me unintimidated encouragement and support in various ways.

I gratefully acknowledge Markus Hidell for his advice, and crucial contribution, which made him a backbone of this research and so to this thesis. His involvement with his originality has triggered and nourished my intellectual maturity that I will benefit from, for a long time to come. Markus, I am grateful in every possible way.

Many thanks go in particular to Tariq Mahmood and Adil Yaqoob. I am much indebted to Tariq Mahmood for his valuable advice in thesis writing and furthermore, using his precious times to read this thesis and gave his critical comments about it.

Words fail me to express my appreciation to my family for their love, support, persistent confidence and most importantly prayers during my studies far away from them.

Finally, It is pleasure to thank all my friends specially Ishrat Ali Awan, Asim Shahzad, Waqas Ghuman and Talha Bin Fida, who supported me during my studies and stay in Sweden. And as well as expressing my apology that I could not mention personally one by one.
Contents

Declaration of Authorship i

Abstract ii

Acknowledgements iv

List of Figures viii

List of Tables x

Abbreviations xi

1 Introduction 1
  1.1 Motivation ......................................................... 1
  1.2 Problem Statement .................................................. 2
  1.3 Thesis objective .................................................... 2
  1.4 Contributions ...................................................... 3
  1.5 Thesis outline ...................................................... 3

2 Background 4
  2.1 Video Streaming ..................................................... 4
  2.2 Challenges of Video Streaming ........................................ 6
  2.3 Streaming Protocols ................................................ 7
     2.3.1 Real time Streaming Protocol (RTSP) ........................... 8
     2.3.2 Session Description Protocol .................................... 10
     2.3.3 Real Time Transport Protocol (RTP) ............................ 12
       2.3.3.1 RTP Header .................................................. 13
       2.3.3.2 RTP Profile .................................................. 14
       2.3.3.3 Real Time Control Protocol (RTCP) .......................... 15
     2.3.4 RTCP Services ................................................... 15
       2.3.4.1 RTCP Packet Types .......................................... 16
     2.3.5 Proprietary Streaming Protocol .................................. 17
       2.3.5.1 Microsoft Media Server ...................................... 17
       2.3.5.2 Shoutcast/Icecast Protocol (ICY) ............................ 17
6.5 Conclusion .......................... 69

7 Conclusion and Future Work 70
  7.1 Conclusion .......................... 70
  7.2 Future Work .......................... 71

A DAVS Code 72

B DAVS Client 73
  B.1 Class dependency Diagram .......................... 73
  B.2 Snapshot of DAVS Client .......................... 73

Bibliography 75
List of Figures

2.2 Streaming modes [11] ................................................................. 6
2.3 RTSP Session[14] ................................................................. 9
2.4 Sample SDP File ................................................................. 12
2.5 Encapsulation of RTP Packet [15] ........................................... 13
2.6 RTP Header ................................................................. 13
2.7 RTCP Sender Report Packet [19] ........................................... 16
2.8 IP Multicasting architecture [26] ........................................... 19
2.9 Internet traffic statistics 2008/2009 [41] .................................. 20
2.10 Tree base approach [43] ...................................................... 22
2.11 Mesh base approach [43] ...................................................... 22
3.1 A Scalable Video Server Architecture [57] .................................. 24
3.2 Yima system architecture [58] .................................................. 25
3.3 Elvira video server architecture [59] ........................................ 26
3.4 QuickTime Streaming Server architecture [47] ......................... 28
3.5 Adobe streaming server architecture [46] .................................. 29
4.1 Distribution agnostic video server modules ................................ 32
4.2 Distribution agnostic video server design .................................. 32
4.3 DAVS Layers ................................................................. 34
4.4 DAVS database design ....................................................... 38
4.5 Available streaming engines on DAVS ...................................... 40
4.6 DAVS API script to retrieved the available streaming engines ..... 40
4.7 DAVS API import script for generating stream ID ..................... 41
4.8 DAVS system overview ...................................................... 43
5.1 Flow of a validation script of DAVS API ................................. 49
5.2 Flow of DAVS API Import script ........................................... 51
5.3 Flow of a start script of DAVS API ........................................ 52
5.4 Flow of a deport script of DAVS API ....................................... 53
5.5 Basics behind RPC client server program [78] ......................... 54
5.6 Remote procedure call (RPC) mechanism [79] ....................... 54
5.7 DAVS.x file according to the specification of RPC language ...... 56
5.8 Video server interface invoking DAVS API validate.sh .......... 57
5.9 Video server interface invoking DAVS API import.sh ............. 57
5.10 DAVS Client packages .................................................... 59
6.1 DAVS client interaction ...................................................... 62
List of Figures

6.2 DAVS experiment setup diagram .......................... 63
6.3 DAVS processor utilization ............................... 66
6.4 DAVS memory utilization ................................. 67
6.5 DAVS network interface utilization ....................... 68

B.1 DAVS client Class Dependency Diagram .................. 73
B.2 DAVS client snapshot .................................. 74
List of Tables

2.1 SDP attributes [18] ......................................................... 11
2.2 RTP Payload type [21] ..................................................... 14
6.1 DAVS test bed configuration ............................................. 64
Abbreviations

ATM  Asynchronous Transfer Mode
CODEC  Coder/DECodec
DAVS  Distribution Agnostic Video Server
DTS  Decoding Time Stamp
ES  Elementary Stream
FEC  Forward Error Correction
FPS  Frame Per Second
FGS  Fine Granularity Scalability
HTTP  Hyper Text Transfer Protocol
ICY  Icecast Protocol
IEC  International Electrotechnical Commission
IETF  International Engineering Task Force
ISO  International Organization Standardization
ITU  International Telecommunication Union
MMS  Multimedia Media Server protocol
MPEG  Moving Picture Expert Group
P2P  Peer to Peer
PCM  Pulse Code Modulation
PTS  Presentation Time Stamps
RTP  Real-time Transport Protocol
RTSP  Real-time Streaming Protocol
RTCP  Real-time Control Protocol
RPC  Remote Procedure Call
RTMP  Real Time Messaging Protocol
RTMPE  Encrypted Real Time Messaging Protocol
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDP</td>
<td>Session Description Protocol</td>
</tr>
<tr>
<td>SIP</td>
<td>Session Initiation Protocol</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>TS</td>
<td>Transport Stream</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>URL</td>
<td>Universal Resource Locator</td>
</tr>
<tr>
<td>VOD</td>
<td>Video on Demand</td>
</tr>
<tr>
<td>WMS</td>
<td>Windows Media Services</td>
</tr>
</tbody>
</table>
To my lovely Parents and sweet Sisters . . .
Chapter 1

Introduction

1.1 Motivation

Video has been an important media for communications and entertainment for many decades. Initially video was captured and transmitted in analog form. The advent of digital integrated circuits and computers led to the digitization of video, and digital video enabled a revolution in the compression and communication of video. Recent advances in computing technology, compression technology, and high-speed networks have made it feasible to provide real-time multimedia services over the Internet. Real-time transport of live video or stored video is the predominant part of real-time multimedia.

Video streaming applications relies normally on a client-server model. In order to access the required video, the client machine relies on a client process, which includes a player to visualize the video, and a streaming machine that must be matched to the streaming technology used by the server. Depending on the type of application (i.e., broadcast like TV or on-demand), the server either starts streaming the video upon request or is already streaming it. The player, the video format, and the streaming technology are often tightly coupled with one another making such applications rather stiff to evolve, and also problematic to support across different systems and platforms.

Indeed, the streaming technique adopted to distribute the video should be entirely independent from the media format and the player chosen. Additionally, the idea of peer peer to peer (P2P) streaming, especially for broadcast applications, is proving to overcome the known limitations of IP multicast and is rising as a promising new paradigm for streaming video.

Due to ample research and interest, new multimedia streaming technologies are gaining support and attention. New trends are supporting technologies where end users
do not have to buffer the content [1], thus also reducing issues about digital right management. In parallel, very popular players like Adobe Flash are considering supporting to peer to peer for efficient use of video streaming [2].

However, new multimedia streaming applications are not immune to interaction with the current Internet and client server model like, nor they solve necessarily problems related to QoS, scalability, bandwidth consumption, etc. Different companies and research institutions have shown their interest in experimenting with different streaming technologies to resolve these issues.

To compare distinct streaming technologies in a real world, we have to setup different streaming servers using different configurations, which eventually surge in the cost in terms of servers and their maintenance.

1.2 Problem Statement

The above considerations call for the design and realization of a video server.

"A video server which can be dynamically coupled to different streaming and distribution techniques, making the service independent, or agnostic, to the streaming technique chosen by the client".

1.3 Thesis objective

The objective of a thesis work is to propose a video server in such a way that it does not confine to only one media format and streaming server. Furthermore, to provide a framework to set up a video server with different streaming engines. To achieve this, an application programming interface (API) needs to be developed that can be dynamically coupled to different streaming engines.

Proposed design of a video server needs to be tested in a test bed with different streaming engines to ensure the "engine agnostic" capability of a video server. Moreover, evaluate the different performance measurement parameters, whether the proposed design has any overhead on a hardware resources of the video server.
1.4 Contributions

We have achieved certain goals at the end of this thesis project and these goals lead me to believe that the thesis work is a considerable contribution to the research and development in the field of multimedia streaming specifically in the area of video servers. The proposed designed and implementation of the video server is built under the GPL license, which is not confined to only limited media formats and streaming engines. The main achievement of this project is the design and development of an API, considered to be generic so that it can be associated with any streaming engine to set up a video server to endorse a variety of multimedia media formats.

1.5 Thesis outline

The report is logically structured to provide the reader with suitable background knowledge before plunging into the details and implementation of a proposed video server. This report is organized in seven chapters as follows:

- **Chapter 2** presents the streaming concept and traditional architectures that have been employed to deliver multimedia contents to end users. We also present the peer-to-peer architecture and how multimedia contents are distributed in different P2P networks.

- **Chapter 3** provides the related work that has been done in development of video servers and discusses some commercial video servers that are currently in the market. Lastly, we conferred the motivation behind our proposed design of a video server.

- **Chapter 4** presents a detailed design of the proposed video server. Different modules are proposed, to bring in an engine agonistic capability in the video server is discussed in detail.

- **Chapter 5** discusses the challenges and implementation details of the distribution agnostic video server (DAVS), and the tools used in development of a video server.

- **Chapter 6** presents the testing of the "engine agnostic" functionality of the video server, when it was deployed in a test bed.

- **Chapter 7** contains conclusions and suggestions for future work.
Chapter 2

Background

In this chapter we present the concepts of multimedia streaming and the protocols that has been evolved over the years to dispatch multimedia contents over the Internet. Traditional architectures such as IP multicast and application level multicast (ALM) will be elaborated as well. Next, peer-to-peer systems are presented. The reasons for adopting a peer-to-peer architecture for live multimedia streaming via the Internet will be presented.

2.1 Video Streaming

The concept of streaming media came at a time when basic multimedia technologies had already established themselves on desktop PCs. Audio and video clips were digitized, encoded (e.g., using MPEG-1 compression standard [3]), and presented as files on the computer’s file system. To view the information recorded in such files, PC users ran special software designed to decompress and render\(^1\) them on the screen. The first and the most natural extension of this paradigm on the Internet was the concept of downloadable media. Compressed media files from the Web were likely to be downloaded on local machines, where they could be played back using the standard multimedia software. However, this was not a acceptable solution for users with limited amounts of disk space, slow connection speeds and/or limited patience. This essentially created the need for streaming media, a technology that empowered the user to experience a multimedia presentation on-the-fly, while it was being downloaded from the Internet.

\(^1\) Rendering is the process of generating an image from a model, by means of computer programs.
that consumes the stream in real-time. This implies that the client must consume the stream at the same rate at which the stream is sent by the server (that is to say, client and server must be synchronized).

![Figure 2.1: Downloading video][1]

In downloading, the entire file is downloaded to the user’s machine before he or she can play a single frame, as depicted in Figure 2.1. In the downloading scenario, a standard web (Hyper Text Transfer Protocol (HTTP) [4]) serve can be used to serve the media file.

Downloading video is no different than downloading any multimedia file from the Web. Clicking on the link or entering the URL (Universal Resource Locator) sends an HTTP request to the server, which then commences the transfer of the file to the user hard disk. After downloading, player software plays the file from the user hard disk.

Actually, most media players have the capability to play the file while it is downloading, as long as it downloads fast enough. However, if the video bit rate is too high for the user’s bandwidth, he may have to wait until it fully downloads before it can play back.

In streaming, a streaming server is used to dispatch chunks of the file to the end user. As soon as a few frames are received, the media player can start playing. As new frames are received, they are stored in a buffer (a section of a memory or disk space), displayed at the appropriate time, and then discarded. New video is pulled via the network to keep this buffer full. The whole process of streaming is illustrated in Figure 2.2 below.

Streams are server based content, meaning that all the video is kept on the streaming server, which only downloads a few frames of video at a time. The player does not (permanently) save the video to the hard disk, when the stream is finished there is nothing left on the hard disk to watch. Streaming from a media server to media client...
allows fairly instant viewing and also allows viewers to skip around within video and give VCR-like controls, by sending commands back to the media server [10].

2.2 Challenges of Video Streaming

Dissemination of real-time video has bandwidth, delay, and loss requirements. However, there is no quality of service (QoS) guarantee for video transmission over the current Internet. Inclusion, for video multi cast, the heterogeneity of the networks and receivers makes it difficult to attain bandwidth efficiency and service flexibility. Consequently, there are many stimulating concerns that need to be addressed for Internet video transmission and streaming applications.

- **Bandwidth**

  To achieve admissible presentation quality, dispatching of real-time video contents typically demand a minimum bandwidth. However, the current Internet does not accommodate bandwidth reservation to meet such a requirement. Available bandwidth between two end points in the Internet is generally unknown and time varying. If the sender disseminates multimedia contents faster than the available bandwidth then congestion occur, which induced a packet loss and that cause a drop in video quality. On the contrary, if sender transmits slower than the available bandwidth then receiver produces sub-optimal video quality [12]. Additionally, since conventional routers typically do not actively participate in congestion control [9], immoderate traffic can cause a congestion collapse, which can further degrade the throughput of real-time video [8].

- **Delay**

  In contrast to data transmission, which is usually not subject to stringent delay restraints, real-time video needs a bounded end-to-end delay. That is, every
video packet must arrive at the destination in time to be decoded and displayed. Because real-time video must be played out in a timely fashion, if the video packet does not arrive on time, then it is useless and can be considered lost, because its time slot for being played has been passed [8]. Although real-time video requires timely delivery, the current Internet does not offer such a delay guarantee. In particular, the congestion in the Internet could provoke an excessive delay, which exceeds the delay requirement of real-time video [12].

- **Loss**

  Loss of packets can potentially make the presentation annoying to human eyes, or, in some cases, make the presentation impossible. A number of different types of loss may occur. To combat the effect of loss, video applications typically enforce some packet loss requirements [8]. Specifically, the packet loss ratio is required to be kept below a threshold to achieve adequate visual quality. Although real-time video has a loss requirement, the current Internet does not provide any loss assurance. In particular, the packet loss ratio could be very high during network congestion, leading to critical degradation of video quality. Approaches for error control can be classified into four classes [12]

  - Forward error correction (FEC)
  - Retransmissions
  - Error concealment
  - Error-resilient video coding.

### 2.3 Streaming Protocols

In recent years, audio/video streaming has become a most prominent applications over the Internet [34]. The dominance of multimedia streaming applications will surpass in upcoming years. Considering the current progress in multimedia networks and the improvements in Internet infrastructure such as high speed networks, advancement in the mobile communication and new QoS oriented protocols will take multimedia streaming application into a new horizon.

Dispatching of streaming media contents over the Internet certainly differs from the standard media transfer. Streaming applications have a constraint of timely delivery, so contents must be played out as soon as they are received. As compared to the traditional file transfer over the Internet; where you cannot access the file contents until it is downloaded into your machine. Hence, streaming applications reveal many comforts
to the end users like he can play out a large file instantly and no need to wait for the entire file to be downloaded.

Due to current advancement in the access networks; network capacity has changed dramatically; consequently, made it possible for the end user to experience multimedia streaming applications in an economical fashion. The Internet Engineering Task Force (IETF) [13] has standardized a set of protocols for carrying real time multimedia content over the network. This section deals with the details of these protocols. This section covers the details of the Real-time Transport Protocol (RTP) [5, 19], which is most commonly used protocol to deliver real time multimedia contents to the end user over the Internet. Moreover, RTP has a lightweight companion protocol called the Real Time Control Protocol (RTCP) [19], whose main purpose is to monitor the QoS of the RTP packets, is also discussed in detail. The Real Time Streaming Protocol (RTSP) [11], which gives VCR like capability during multimedia session is also discussed. We also describe the Session Description Protocol (SDP) [18] and other streaming protocols such Shoutcast/Icecast (ICY) and Microsoft media server protocol (MMS).

In video streaming, client request compressed video files, which are residing on servers. Upon receiving the client request, server directs a video file to the client by sending the file into a socket (Both TCP [6] and UDP [7] socket connections are used in practice.). Before sending the video file into the network, file is segmented, and the segments are typically encapsulated with RTP header. RTP aims to provide services suitable for the transport of real time media, such as audio and video, over an IP networks. These services accommodate timing recovery, loss detection and correction, payload and source identification media synchronization etc. User interactivity between client and server such as play, pause, and stop etc. is accomplished through IETF standard Real time Streaming Protocol (RTSP).

2.3.1 Real time Streaming Protocol (RTSP)

IETF has standardized a protocol in RFC 2326 [14], called Real Time Streaming Protocol (RTSP), which provides ‘VCR-like’ functionality for audio and video streams, like pause, fast forward, reverse and absolute positioning. RTSP is an application-level protocol designed to work with lower-level protocols like RTP, RSVP to provide complete streaming services over Internet.

RTSP is a client server multimedia presentation protocol to empower controlled delivery streamed multimedia data over IP network. It renders to entail for opting delivery channels such as UDP, multicast UDP and TCP, and delivery based upon RTP. RTSP specification not only endorses for single viewer unicast but also support for large
multicast audience. Sources of data can incorporate both live data feeds and stored clips [15, 16].

RTSP is an out-of-band protocol, meaning that it is not part of the stream itself. It is usually carried over TCP, using a default port of 554. In RTSP specification, presentation refers to a set of streams belonging together, and treated as a single entity by the client. One of the simplest examples would be a presentation comprises of both audio and video stream. Both presentations and single streams are identifies by RTSP URLs (rtsp://<address>/<session>/<stream>) [14, 17].

RTSP has been deliberately designed to provide same services on streamed audio and video just as Hyper Text Transfer Protocol (HTTP) does over the Internet. RTSP has a similar syntax and operations so that extension mechanism to HTTP can also be added to RTSP [14].

However, RTSP differs in many aspects from HTTP. HTTP is a stateless protocol, while RTSP is a stateful. RTSP server keeps state information for each client as long as the connection is open. HTTP is an asymmetric protocol, where client issues request and the server response, but in RTSP both media server and the client can issue request [14]. Another big difference respect to HTTP is that HTTP performs both signaling, control, and transport of the media stream, while RTSP generally provides only signaling and control (the media stream is generally transported over RTP). Figure 2.3 demonstrates a possible interaction between client and server below.

The most important RTSP commands are [14]:

**OPTIONS**: Client or the server can issue this command at any time to assure other party the available commands it can accepts.
SETUP: Client asks the server to allocate resources for a stream and start a RTSP session.

ANNOUNCE: require information about the available media content

DESCRIBE: retrieves the description of a presentation or media object identified by the request URI from a server. (The returned packet will embed an SDP)

PLAY: tells the server to start sending data via the mechanism specified in the SETUP method.

RECORD: This method initiates recording a range of media data according to the presentation description. The timestamp reflects start and end time (UTC).

In some cases, the media stream cannot be controlled by the client, and only signaling has to be performed (this is often the case with push streaming). In this case, HTTP and RTSP can still be used: for example, the SDP describing the stream can be published on a Web page, and the client can download it through HTTP to watch the stream. Or RTSP can be used to get the SDP (in this case, the only important RTSP command is DESCRIBE).

Also note that RTSP URL is often published in Web pages. If the client is not provided with a return channel (that is, if the network connection is unidirectional from the server to the client), signaling cannot be performed using request-based protocols such as HTTP or RTSP. In this case, the session description can be distributed off-line, or signaling can be performed using a unidirectional protocol such as SAP. The idea behind this kind of protocols is to use multicast traffic for distributing the SDP (note that since the network is unidirectional, the server does not know the clients’ addresses. So, it has to use multicast for distributing the SDP).

2.3.2 Session Description Protocol

A media session requires certain parameters to be known in advanced; in order to a successful media session. These parameters include ports, addresses, codecs used by the participants, description of the actual streams etc. IETF has standardized a protocol defined in RFC 2327 [18]; which provides a procedure for describing session parameters that would be used in a media session between two participant. SDP specification only defines to describe sessions, not streams. A SDP session may contain several media streams. A SDP did not define how these parameters would transport; which implies that it can be carried by numerous transport and application protocols. Hence it can be published on a webpage, embedded into a RTSP or SIP message, or even sent via email.
SDP session is textual protocol and session descriptions are entirely textual using ISO 10646 character set in UTF-8 encoding. SDP session description comprises of a number of lines of text of the form \(<\text{type} > = <\text{value} >\). \(<\text{type} >\) is always exactly one character long and is case-sensitive. While \(<\text{value} >\) is a structured text string whose format is depends on \(<\text{type} >\) value and important \(<\text{type} >\) values are shown in Table 2.1 below [18]. The order of the \(<\text{type} >\) attribute is strictly define in RFC 4566, in order to allow detection of error messages and rapid procession. The type fields can be divided into three categories; first category describe the session, second provides the information about when and how long the session will be active and last describes the media that is carried in the session. An SDP session description includes the following media information such as the type of media, transport of protocol (RTP/UDP, H.320, etc.), format of the media (H.261 video, MPEG video etc.). Apart from conveying media related information to other party, it also describes the address and port details.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>Protocol version</td>
</tr>
<tr>
<td>o</td>
<td>Origin</td>
</tr>
<tr>
<td>s</td>
<td>Session Time</td>
</tr>
<tr>
<td>t</td>
<td>Time</td>
</tr>
<tr>
<td>c</td>
<td>Connection data</td>
</tr>
<tr>
<td>m</td>
<td>Media name and transport data</td>
</tr>
<tr>
<td>a</td>
<td>Media attribute</td>
</tr>
</tbody>
</table>

Table 2.1: SDP attributes [18]

Like in multicast case SDP contains the address of the multicast group address and transport of media and for unicast IP session SDP file contains information of the remote address for media and remote transport port of media [4]. Figure 2.4 exhibits a sample SDP file below.

In this example, the first 7 lines globally describe the session, while the remaining 4 lines describe two media streams (a video stream and an audio stream). The ‘v=0’ line indicates the SDP version (0), whereas the ‘o=...’ line describes the creator of the session described by this SDP file, ‘i=...’ line provides some additional information regarding such session, and the ‘t=0 0’ line indicates the time when the session is available. Some ‘a=’ lines can be added to add information that are not taken into account by RFC 2327. The ‘m=...’ (media name and transport address) and ‘c=...’ (connection information) lines describe the two streams: for example, m=audio 6666 RTP/AVP 14 indicates that the stream on UDP port 6666 is an audio stream transmitted over RTP, with payload type 14 (note that the static payload type 14 is associated to MPEG1 audio by RFC 1890), and the c=IN IP4 239.255.42.42/127 indicates that the stream is sent over IPv4, using the multicast group 239.255.42.42, with time to live (TTL) 127.
2.3.3 Real Time Transport Protocol (RTP)

IETF has standardized a protocol called Real-time Transport Protocol (RTP), defined in RFC 3550, which carry multimedia traffic over an IP network. Applications transmitting real-time data, such as audio, video or simulation data, over unicast multicast network; RTP provides end-to-end network transport functions. RTP designed in such a way to work with its companion control protocol, Real Time Control Protocol (RTCP), which acquires feedback on quality of multimedia content transmission in an on going session. Dispatching of multimedia contents amplify by RTCP, which provides monitoring of multimedia media content delivery in a scalable way to large multicast networks, and to provide minimal control and identification functionality [19].

According to RTP specification, it provides end-to-end delivery services for real-time multimedia content such as payload type identification, sequence numbering, time stamping, loss detection. RTP is primitively designed to conform to the requirements for multicast of real-time data. Since that time, it has proven useful for a wide range of other applications such as web casting, video conferencing, and TV distribution and in both wired and cellular telephony [20].

RTP usually runs on top of UDP to employ its multiplexing and checksum functions. Over the Internet, TCP and UDP are two most well known transport protocols. TCP provides connection-oriented and reliable communication between two hosts, while UDP provides connectionless and unreliable datagram service over the network. UDP was preferred as the transport protocol for RTP because TCP does not scale well and dominant feature of TCP is reliability, which is not required. In multimedia communication, significance of reliability is not as important as timely delivery of the real time
multimedia contents. Figure 2.5 shows the RTP packet encapsulated in an IP/UDP packet.

![Figure 2.5: Encapsulation of RTP Packet [15]](image)

### 2.3.3.1 RTP Header

The format of RTP header is illustrated in Figure 2.6 below. The first twelve octets are present in every RTP packet, while the list of CSRC is present only when inserted by a mixer [19, 20].

![Figure 2.6: RTP Header](image)

**Version (2 bits)**: This field identifies the version of RTP. Currently version in use is 2.

**Padding (1 bit)**: If the padding bit is set, then packet contains one or more additional padding octets at the end which are not part of the payload.

**Extension Header (1 bit)**: If the extension bit is set, the fixed header must be followed by exactly one header extension.

**CSRC count (4 bits)**: Under normal scenario, RTP data is generated by a single source, however when multiple RTP streams pass through a mixer\(^2\) or translator, multiple data sources may have contributed to an RTP data packet. CSRC count contains

\(^2\)A mixer is an intermediate system that receives RTP packets from a group of sources and combines them into a single output, possibly changing the encoding, before forwarding the result.
the number of CSRC identifiers that followed the fixed header.

**Market (1 bit)**: The marker bit in the RTP header is used to mark events of interest within a media stream; its precise meaning is defined by the RTP profile and media type in use.

**Payload Type (7 bits)**: This field identifies the format of the RTP payload and tells the receiving application the media type that is transported in this packet. The mapping of payload type and media formats can be done statically by RTP profile or dynamically through signaling mechanism such as SDP [19]. Table 2.2 lists some of the video payload types currently supported by RTP.

<table>
<thead>
<tr>
<th>Payload type number</th>
<th>Video format</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>Motion JPEG</td>
</tr>
<tr>
<td>31</td>
<td>H.261</td>
</tr>
<tr>
<td>32</td>
<td>MPEG1 video</td>
</tr>
<tr>
<td>33</td>
<td>MPEG2 video</td>
</tr>
</tbody>
</table>

Table 2.2: RTP Payload type [21]

**Sequence Number (16 bits)**: The primary purpose of sequence number in RTP packet is to detect packet loss and out of order delivery mainly caused by the underlying network. The initial value of the sequence number should be random and increments by one for each RTP packet send.

**Time stamps (32 bits)**: This field is employed so that the receiver can reconstruct the payload’s position in the session timeline (i.e., its relative temporal base). The first media sample is assigned a random timestamp and all subsequent packets add a payload-dependent offset to this value.

**SSRC (32 bits)**: The SSRC field identifies the synchronization source which identifies the source of the transmission. This identifier should be chosen at random so that no two synchronization source within the same RTP session will have same SSRC identifier.

### 2.3.3.2 RTP Profile

Basic RTP header often contains insufficient information for the client to interpret the contents of the packet correctly. RTP design intentionally in this way, because including all the data necessary for all possible media formats would make the header muddled and waste a lot of bandwidth. RTP can be extended via profiles and payload format description to append media dependent information.

RTP profile in use today is the "RTP profile for Audio and Video Conferences with Minimal Control (RTP/AVP or AVP)" [7]. The profile does little more than provide
guidelines regarding audio sampling, slightly relaxing RTCP timing constraints, and defining a set of default payload type/media format mappings [20].

2.3.3.3 Real Time Control Protocol (RTCP)

The Real-time Transport Control Protocol (RTCP) is a companion protocol to RTP and is defined in the same RFC as in RTP [19]. The design principle behind the RTCP is to provide feedback in an ongoing session regarding the quality of the session to the participants. In an RTP session, participants periodically send RTCP packets to deliver feedback on quality of content delivery and information of membership.

2.3.4 RTCP Services

In [19] defines that RTCP provides following four services.

- **QoS monitoring and Congestion Control**
  The primary function of RTCP is to provide the feedback to an application regarding the quality of data distribution. The feedback is in the form of sender reports and receiver reports. Sender reports send by the sender; receiver reports send by the receiver. Reports comprises the information related to the quality of reception such as fraction of lost RTP packets, since the last reports; accumulative number of lost packets, since the RTP session begins; delay since receiving the last sender’s reports etc.. RTCP feedback really helpful for the sender and receiver both, sender can adjust its transmission rate; receiver can determine whether congestion is local, regional or global.

- **Source Identification**
  In RTP data packets, sources are identified by randomly generated 32-bit identifiers called SSRC. Unfortunately, SSRC identifies is not convenient for human users. RTCP provides a human friendly mechanism for source identification, to remedy this issue. RTCP SDES (source description) packets contain textual information called canonical name (CNAME) as globally unique identifiers of the session participants. It may include a user’s name, telephone number, email address and other information [23].

- **Control packets scaling**
  It is specifies in [19], RTCP packets are send periodically among participants. However, when the number of participants increases, there should a balance between getting feedback of the participants in a RTP session and the limiting control traffic. RTP limits the control traffic to at most 5
• **Inter media synchronization**

RTP sender reports contain an indication of real time and the corresponding RTP timestamp. This can be used in inter-media synchronization like lip synchronization in video.

### 2.3.4.1 RTCP Packet Types

RFC 3550 defines 5 packet types of RTCP, which are described below.

- **Receiver Report (RR):**
  Receiver reports contain reception quality feedback about content delivery, including the highest number of packets received, the number of packets lost, inter-arrival jitter and the time stamp to calculate the round-trip delay between the sender and receiver. These reports are generated by the receiver.

- **Sender Report (SR):**
  Sender reports are generated by active senders. The main purpose of these reports is to aid the receiver in synchronizing multiple media streams for instance audio and video. The structure of an RTCP SR packet is described in Figure 2.7 in particular; the NTP timestamp is common to all the streams of all the sessions, while the RTP timestamp field is in the same temporal unit of the timestamps contained in the RTP packets of the same stream. So, these two fields can be used to map the timestamps contained in RTP packets to absolute NTP timestamps that can be used to synchronize different media streams. As a consequence of this mechanism, it is not possible to play synchronized audio and video until the first SR packet has been received for all the streams. As a consequence of this mechanism, it is not possible to play synchronized audio and video until the first SR packet has been received for all the streams.

<table>
<thead>
<tr>
<th>V</th>
<th>P</th>
<th>RC</th>
<th>PT</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSRC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NTP Time stamp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTP Time stamp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 2.7: RTCP Sender Report Packet** [19]
Chapter 2. Background

- **Source Description (SDES):**
  SDES is used to transmit information about the user to other participants in the session. The canonical name (CNAME) item is present in each SDES and is used to identify a participant across sessions.

- **RTCP BYE:**
  RTCP BYE is sent to notify other participants that user is leaving the session.

- **RTCP APP:**
  RTCP APP is an application-dependent extension. It is basically now on experimental use for future application.

### 2.3.5 Proprietary Streaming Protocol

#### 2.3.5.1 Microsoft Media Server

Microsoft Media Server (MMS) [24] protocol is a Microsoft proprietary protocol used to transfer unicast data in Windows media services. In the late 1990s, Microsoft developed its own set of protocols for media delivery, although they already employed RTP in their Net meeting conferencing application.

Microsoft developed MMS protocol, which integrated most of the features of RTP, RTCP and RTSP. To attain the attention of widest possible audience, Microsoft designed their protocol with several different versions, each going over a more restricted kind of network [24].

- MMSU goes over UDP for the most efficient delivery.
- MMST goes over TCP for networks that do not permit UDP traffic.
- HTTP carries the MMS protocol over HTTP for networks that allow only HTTP traffic due to firewalls.

#### 2.3.5.2 Shoutcast/Icecast Protocol (ICY)

A company called Null soft (now part of AOL), using a slightly customized version of the HTTP protocol called ICY protocol with a URL like icy://www.mydomain.com:8200, created the Shoutcast server, which can send or receive streamed MP3 or pretty much any streamable audio or video codec.[25].

Shoutcast consist of a client server model and Shoutcast servers and clients are available for Palm OS, Microsoft Windows, FreeBSD, Linux, Mac OS X and Solaris...
The cost of setting up own broadcasting network is very minimal as compare to traditional AM broadcasting or FM radio station. So some traditional radio stations make use of Shoutcast service to extend their presence onto the web.

### 2.4 Multicast Streaming

In recent years, the dramatic growth of Internet users and their interest in the video streaming applications has shown that a single server is not being able to server large amount on Internet audience, despite of the fact that tremendous progress has been made in the improving the performance of software and hardware of streaming media servers.

Furthermore, single server base delivery system faces several major problems from network utilization point of view. The amount of traffic it pushes to its clients is always a linear function to the number of subscribed clients [40]. This generates a large quantity of network traffic which causes the network congestion. Media server sends data to each individual client, even if the content is same. To circumvent this problem, multicast routing is deployed in the network, which reduced the load on media server. In this way, server only sends a single stream, and if network support multicast, it duplicate the content and sends the content where clients are available and demand the contents.

In Multicasting, packets are sent from one sender to many other receivers without unnecessary packet replication in an IP network. In multicasting, one packet is sent from a source and is replicated as needed in the network to reach as many end-users as necessary. In networking jargon, multicasting is not the same as "broadcasting": broadcast data are sent to every possible receiver while multicasts are sent only to those receivers that shown their interest for that particular data. Figure 2.8 depicts the IP multicasting mechanism.

Over the years, many schemes have been proposed in the routing architecture to support the multicast transmission. These schemes made possible for the transmission of multicast data over the existing IP infrastructure. In [27, 28], Deering proposed a scheme for multicast routing architecture, which are implemented in Multicast open shortest path first (MOSPF) [29] and Distance vector multicast-routing protocol (DVMRP) [30].

The proposed schemes were employed within a region, where the availability of bandwidth is enough to support the multicast services. However, when group members are distributed sparsely across a wide area, these proposed schemes are not efficient. In case of DVMRP, data packets and in MOSPF membership information reports are
Chapter 2. Background

Figure 2.8: IP Multicasting architecture [26]

send on links, and associated state is stored in routers, that do not lead to receivers or senders, respectively [22].

Multicast modes fall into two categories. Dense multicast (such as Protocol Independent Multicast Dense Multicast, PIM-DM [31]), and sparse multicast (such as Protocol Independent Multicast Sparse Multicast [32], PIM-SM). PIM-DM is designed for the multicast LAN applications, while PIM-SM is for wide area, inter-domain network multicasts. End user or hosts usually employed the Internet Group Membership Protocol (IGMP) to join or leave a multicast stream [33]. IGMP is the control mechanism used to control the delivery of multicast traffic to interested and authorized users.

Unfortunately, Multicast routing prevail many benefits to the operators but it is very common over the Internet that operators did not support of multicasting. Currently IP Multicast has scalability problems, when large number of users and groups reached, it is not yet accepted globally for being a solution to Internet wide multicast [33]. This motivates the development of so-called Application Level Multicast (ALM) networks that employs multiple intermediate servers that re-broadcast packets to the respective clients [10, 40].

2.5 Peer to Peer Streaming

Providing cost effective multimedia streaming services on large scale has been always a subtle problem. Over the years, the growth and popularity of peer-to-peer system has been tremendous. Due to the popularity and acceptance of peer-to-peer multimedia file sharing applications such as Gnutella [35], Napster [36] and Kazaa [39] over the Internet from the last decades, have made possible to embrace this into multimedia streaming. Recently, peer-to-peer system has emerged a promising technique to deploy multimedia
streaming services. This new paradigm brings many advantages which traditional client server model is lacking over the Internet such as scalability, resilience and effectiveness to cope with dynamics and heterogeneity. A recent study of Internet traffic shows that P2P traffic is dominating over the Internet [41]; as shown in Figure 2.9 below.

![Distribution of protocol classes 2008/2009](image)

**Figure 2.9: Internet traffic statistics 2008/2009 [41]**

Unfortunately, streaming servers often get overloaded, when large number of user’s requests hits the server and because of that video quality degraded. This is where P2P technology can help to remedy this problem. An experiment study [45] shows that; peer to peer would be a viable alternative to the traditional client server architecture.

Peer-to-peer systems can be classified into two categories. The first category of peer-to-peer systems is based on their degree of centralization. Other is based on their structure; such as structured (tree base approach) and unstructured (mesh base approach) [44].

In [51], peer-to-peer systems can be classified into two categories based on the degree of centralization. One is pure peer-to-peer system and other is hyprid, which merge the characteristics of client server architecture and peer-to-peer system.

In Pure peer-to-peer systems, there is no need for central entity for managing the network. Peers are treated equally and each of them provides the functionality of both client and server. Gnutella [35] is an example of pure peer-to-peer system. In [35], there is no central database that stores the information of all the available files over the Gnutella network. Rather, Gnutella employs a distributed query approach to search a file over the network.
In hybrid approach, characteristic of client server and peer-to-peer system is being merged. **BitTorrent** [37] and **Kaaza** [39] falls in this category. In **BitTorrent**, only information about the file is kept on the special server called **tracker**. Each user connects to the tracker and gets the appropriate meta information. With this meta information, user starts downloading the file from the sources specified in it. A special type of hybrid peer-to-peer systems is being introduced by the **Kaaza** [39]. Kaaza introduces a special type of nodes called a **Super Peers**. **Super Peers** contains some extra information, which other normal peers may not have. If normal peers cannot find the information they are looking for, they contact the super peers for the information. Next section will give you a brief understanding of the classification of the peer-to-peer systems based on its structure.

### 2.5.1 Structured approach

In structured systems, peer formed links with each other and create a topology like of trees or graph as shown in Figure 2.10 below. The challenging task is to create and maintain the topology. Once the topology is formed; the discovery process and downloading is very quick. However, complications begin, when peers join and leave frequently or unexpectedly. When peer is anticipating leaving the system, then tree grafting operations are performed and a whole structure is updated. But when peer leaves unexpectedly then the structured should be destroyed and built it from beginning. Such approaches are typically called ”**push base**”, and the relationship among peers in this approach is well defines like ”parent child” relationship in trees. Hence, when a node receives a data packet, it also forwards copies of the data to each of its children.

Tree base approaches are perhaps the most natural approach. However one pertains to tree-based approaches is that the failure of nodes particularly higher in the tree may disturb the delivery of data to large number of users, as a consequences poor transient performance. In tree base approach, uploading bandwidth of majority peers and its resources are not fully utilized. Researchers have been investigating to over come these issues and working on more resilient structures for data delivery. One approach that gained much attention is called multi tree based approaches [42].

In multi tree streaming, server splits the stream into multiple sub streams, where as single streaming tree is constructed in tree base approach (single tree base approach). Now instead of one streaming tree, multiple sub trees are constructed, one for each sub stream. Each peer joins all sub trees to retrieve all sub streams [43].
2.5.2 Unstructured approach

In tree base system, there is draw back of single point of failure. If a peer’s parent leaves, the peer as well as its descendants cannot get streaming until he joins to another parent. As discussed previously, in tree base approach the management of a streaming tree topology is a challenging task because of frequent peer arrival and leaves. In mesh base approach, there is no static streaming topology as shown in Figure 2.11 below. Relationships among peers are established and terminate dynamically. At any given time, a peer maintains peering relationship with multiple neighboring peers. In mesh base approach, a peer can download/upload video from/to multiple neighbors simultaneously. If a peer’s neighbor leaves then he can still continue downloading from remaining neighbors.
Chapter 3

Related Work

From the last two decades, research community from both the academic and industry have shown their concern and contributed a lot in the field of multimedia communication. Due to their efforts, a wide range of possible future multimedia applications for commercial, educational purposes have been evolved. With the advent of high speed broadband networks, end users are experiencing these new multimedia applications.

All these multimedia applications such as video on demand (VOD), live broadcasting, distance learning, etc. are craved of storage capacity, storage bandwidth, and transmission bandwidth. And most significantly, these multimedia applications required that multimedia contents should be delivered to the end user in a timely fashion. Today there exist wide ranges of multimedia streaming applications, which demand different size of video servers. To support these multimedia applications video servers should be scalable and flexible. Due to varying nature of effective bandwidth and packet loss over the Internet, several strategies have been advised and implemented for content rate adaptation.

Albeit, numerous proposals have been contrived on various aspects of digital video server design, but there has been little work that actually renders us to the design and implementation of a video server that can endorse different streaming engines.

In this section, we will present and describe these contemporary streaming servers and their architectures. We are going to restrict our discussion to only streaming servers since our system is planned for video streaming only. We are going to depict not only proposals originating from the research community, but also the commercial streaming servers that are becoming the industry trends in the market.
Christoph B. [57] proposes a new architecture where video servers are configured into a server array just like a disk array. On the contrary, to the single node, that conventionally called as an autonomous and independent machine that does the streaming; It does not store an entire stream. Instead a single stream is distributed over several nodes of the proposed server array. Each server node only stores a sub stream of the original stream. The proposed architecture enforces a restriction that the end user must be able to deal with the multiple sub streams and potentially capable of a fragment the stream into multiple sub streams. Figure 3.1 illustrate the scalable video server architecture.

Figure 3.1: A Scalable Video Server Architecture [57]

Cyrus S. [58] describe that, most of the available media servers present in the market fall into two categories when the solution is a low cost and only single node served only a limited number of users. These systems are typically called consumer-oriented systems. RealNetworks, Apple Computer, and Microsoft product offerings fall into this family. However, other is multi node, carrier-class systems such as high end broad casting and dedicated video-on-demand systems. They designed and developed a new continuous media server which they called 'Yimma'. It uses IP network and a complete end-to-end system that support many clients. Yimma distinguished it from previous research work that on all nodes runs identical software and removes the single point of a failure threat from the system, efficient online scalability by allowing disks to be added or removed without disturbing continuous streams, synchronization among streams, independence from media types, compliance with industry standards, multi threshold buffering flow-control mechanism to endorse variable bit-rate (VBR). Figure 3.2 illustrate the Yimma video server architecture.
Olav S. in [59] designed a video server which is so called Elvira’s video server. The approach they used in designing the Elvira video server is to build a cluster of UNIX workstation interconnected by a high speed ATM switch. The design of the Elvira video server yields several advantages like scalability, by adding more workstations if the demand increases and other is to deploy a simple hardware and software on a machine because the Elvira’s video server is a cluster of workstations. Elvira’s video server supported different allocation strategies, one is to divide a video file into small segments and distribute on server nodes across the server in a round robin fashion and other is to store a video file on one node in the video server. The principal motivation behind this video server is to set up experiment with these strategies. Figure 3.3 illustrate the Elvira’s video server architecture.

Normally, multimedia contents are distributed over the Internet using RTP, but in near future HTTP base streaming will play a big role in the delivery of video contents over the Internet [61]. As the name implies, HTTP, which is used to transfer web contents over the internet, is used to dispatch multimedia contents. In HTTP base streaming, the media file is downloaded as ordinary web pages, but play out the file begins instantaneously as first bytes are received. This approach is employed by the most popular video sharing sites over the internet like YouTube, MySpace, Vimeo etc. [65].

Figure 3.2: Yima system architecture [58]
The popularity of HTTP base streaming urge giants like Microsoft and Akamai to bring together. Joint venture of both the companies in October 2008 takes multimedia streaming into a new horizon [60]. They introduce a new service, which enables the end user to experience a high definition (HD) video streaming, while other users whose connection bandwidth is limited, will experience the streaming according to its connectivity.

Microsoft introduced a new web server technology call Internet Information Services 7.0 (IIS) [62] smooth streaming, Smooth streaming is a new extension for IIS 7.0 media services, enabling adaptive streaming media to Silverlight client [63] through HTTP. Smooth streaming [64] dynamically observes the local bandwidth and CPU conditions in a real time and based on their current conditions, it seamlessly switches the video quality of media that Silverlight player plays.

BT patent streaming architecture which is so called Fastnet [67] is a high performance, network agnostic video streaming architecture that exclude all the well known issues from the streaming systems such as slow start, picture freezing on network congestion and poor video quality. In Fastnets, video is encoded into multiple streams at different data rates like video is encoded at 30Kbps and same video is possibly encoded at 18Kbps and 8Kbps for lower quality. Instant start is achieved by encoding a video into multiple data rates. First sever dispatch the 30Kbps stream. However, if server determines that player is not experiencing data at this rate, then server switches to the 18Kbps. This can be done without any interruption. If later server detects that throughput is improving, then server switch back to higher stream again without pause in video playback.
3.1 Commercial Streaming Servers

There are several streaming servers that are currently available in the market. This section mentions some of them and presents their architecture.

3.1.1 Real Network’s Helix Server and Proxy

Real Networks’ streaming solution, the Helix Server, is a software solution for dispatching multimedia streaming content (audio and video) to PCs or mobile devices. Helix Server is the only multi-format, cross platform streaming server for delivering the highest quality experience to PCs and mobile devices. It is the only digital server that endorses both live and on demand delivery of multiple formats such as Real Media, Windows Media, QuickTime, MP3, H.264, AAC and more [48]. It supports more platforms than any other major streaming server including Linux RHEL 4.0, Windows 2003 and native support for Solaris 10. By caching and splitting content closer to the end-user, Helix Proxy reduces transmission problems that degrade the quality of the playback experience. Helix also authenticates every client request at the start by masking the IP address of your internal users, thereby controlling content and increasing content security.

3.1.2 Apple’s and QuickTime’s Darwin Streaming Servers

The Darwin Streaming Server [49] is an open-source version of QuickTime Streaming Server. The Darwin server allows us to disseminate multimedia streaming contents to the end users using the industry standard protocols such as RTSP, RTP across the Internet. As it is an open source, so it furnishes us the great level of customizability, and allowing us to manipulate the code according to our needs. Darwin streaming server is developed to support multiple platforms such as Linux, Windows and Solaris.

QuickTime Streaming Server (QTSS) [50] is Apple’s commercial streaming server delivered as part of Mac OS X Server. QTSS provides users with enhanced administration and media management tools as a result of the tight integration with Mac OS X Server. These administration and management tools are not part of the open source project.

Figure 3.4 illustrate the QTSS streaming server architecture. The Streaming Server consists of one parent process that forks a child process, which is the core server. The parent process waits for the child process to exit. If the child process exits with an error, the parent process forks a new child process. The core server acts as an interface between
network clients, which use RTP and RTSP to send requests and receive responses, and server modules, which process requests and send packets to the client [47].

![Figure 3.4: QuickTime Streaming Server architecture [47]](image)

3.1.3 Flash Media Server

Adobe Flash Media Streaming Server software [52] is a more secure and affordable step up from using a progressive download to deliver video. End user can now experience a superior video quality with new features that has been added in the system such as dynamic streaming [53], which ensures to expeditiously dispatch of multimedia contents by dynamically switching among different streams with varying quality and size. The whole procedure should be smooth and seamless to the end user. HTTP delivery support and H.264 enhancements that allow for smooth streaming on the web to the Adobe Flash Player runtime, to mobile phones with Adobe Flash Lite™3 software, and to Adobe Media Player software.

Flash media server [54] communicates with its clients by employing the Adobe patent Real Time Messaging Protocol (RTMP [55]) over TCP. The communication channel between the server and client is a two way, which allow the server to send audio, video and data between client and server. Flash media server also offers us an option to set up a secure communication channel between client and server by employing the Encrypted RTMP (RMTPE), which is easy to deploy and empower us to set up a strong stream security by employing RTMPE than using SSL.

Communication between the client and the server is done over a persistent connection using Real-Time Messaging Protocol(RTMP). The client creates a socket connection to Flash Media Server over RTMP. The connection allows data to stream between client and server in real time. Figure 3.5 illustrate the Adobe Flash media server architecture and the whole process [46].
Chapter 3. Related Work

3.1.4 Microsoft Windows Media Services (WMS)

Microsoft Windows Media Services [56], a streaming media server that allows industry to set up a platform for providing streaming services such as streaming live contents, Video on demand (VOD) etc. It only supports windows media, JPEG and MP3 formats. Windows media server provides streaming services with functionalities that make this product distinguish from other streaming servers. These add on functionalities brings more revenue by attracting not only corporate clients but also the small companies and enterprise who are interested to deploy streaming application within their premises. Windows media server adds many functionalities into their system such as support for caching the media contents which not only reduce operation cost but also improve viewing experience for user, enforce authentication, impose connection limit and support of multiple protocols such as (unicast, multicast).

3.2 Conclusion

Several studies have aimed at improving the performance of a video server by introducing multiple nodes with same configuration over a high speed networks to dispatch the multimedia contents [57–59]. Usually, these systems are configured in such a way to support only one streaming engine and have identical software and hardware configuration. These systems are employed to remedy the problem of single point of failure, when large numbers of user’s are expected to hit the video server. However, there is no single best solution or server, which can incorporate with different streaming engines.
Chapter 4

Distribution Agnostic Video Server (DAVS) Design

In this chapter, we elaborate on the architectural components of the video server and their properties. In section 4.1 we provide a general overview and justify the important architectural decisions. In section 4.4, we explain in greater detail the functionalities of the video server components, as well as, point out their advantages design-wise. Finally, the whole DAVS system described in detail in section 4.6.

4.1 Design Approaches

The idea behind in DAVS was to speed up the process for setting up different streaming experiments with different streaming technologies. During the design phase, the most challenging part was to bring ”engine-agnostic” capability into the video server. However, video server will be mainly employed by the research communities for the experiment purposes. So design should be flexible enough to support large number of streaming experiments which endorse large number of different multimedia formats.

For disseminating multimedia contents over the Internet, there exists many possible alternatives for designing a video server that performs ”engine-agnostic” functionality. We can opt either the design approach of the server array based video server [57] or the Elvira video server [59], but it will not only surge the cost in terms of number a video servers and also impose an extra overhead of maintenance and distributing the available sub-streams of each stream in available streaming engine over these video servers.

One design approach is to distribute multimedia contents over different video servers that are configured with different streaming engines. Moreover, bring an intelligence in
some node, where all the requests are first directed towards that node and then direct the message towards targeted node that has been configured with the particular streaming technology. However, this approach will not suited for DAVS, because it will increase the cost of deployment and maintenance.

Instead, we came up with a different design approach, we defined set of scripts that bring the ”engine-agnostic” capability into a video server. We combined all the scripts into an API. Moreover, a new set of functionalities has been defined that are supposed to interact with the well defined API. For all such requirements that mentioned above needs to be met, therefore we opted the modular approach and divide the functionalities of a video server and defined them in different modules.

We are more inclined towards modular approach, as it facilitates in many ways during the implementation phase. Moreover it empowers us the flexibility in the design. In future, if any one interested to add up some functionality in the DAVS API or other modules, it will not influence the working and performance of other modules. Modular approach also imparts an ease of management of multimedia content. The reason behind opting a modular approach that is deploys a good programming practices and adds more functionality into the different video server modules like the DAVS API, a base which introduce a ”engine agnostic” functionality into the video server.

4.2 DAVS Design

The video server will be used as a test bed for running video streaming experiments using different streaming technologies and engines. Therefore, the video server must be designed not to depend on a particular streaming technology or on particular streaming software. On the contrary, the video server will implement an ”engine-agnostic” abstraction that will help to automate and repeat deterministic streaming experiments using different engines. Hence, its functionalities and API should be generic enough to accommodate every hypothetical streaming software and technology (example: RTP/RTSP, RTP/multicast - described by an SDP -, HTTP, various different P2P protocols, etc...).

The video server is based on a modular design, and is composed by a set of components, as shown in Figure 4.1:

- Video Server interface (through which users can access the video server)
- One or more Streaming Engines (which are in charge of performing all the ”streaming work”, but is not user friendly),
• Video Database (used by the streaming engine to keep track of the available streams, and their descriptions)

• Audio/Video Encoder/Transcoder (which can encode audio and video on the fly, or transcode media streams so that the streaming engine can handle them).

![Figure 4.1: Distribution agnostic video server modules](image)

When all these well defined modules are combined together, this will lead us to design a video server which gives us flexibility to plug in different streaming engines and provides us the scalability to perform the experiments on large scale to quantify different parameters in different access networks for example to measure QoS in congested networks, to measure the packet loss in mobile multimedia networks etc. Block diagram of DAVS is exhibited in Figure 4.2 below.

![Figure 4.2: Distribution agnostic video server design](image)

### 4.3 Functionalities

In general, all the possible streaming services provided by the server can be grouped in 2 big categories:

- **Video on Demand (VoD) like services**
  
The server allows some users (the producers) to upload media files that can be
requested and played by clients. Clients can play a stream at any time (for example, by requesting an rtsp://... URL), starting to reproduce it from the beginning. In case of P2P streaming, a producer can simply notify the server that it is sharing a video file, instead of uploading the file to the server. This kind of functionality is roughly equivalent to the one provided by (for example) YouTube.

• **Broadcast-Like services**

The streams provided by producers are available only during a specified time range. The reproduction of the stream starts at the beginning of the time range, and clients that starts playing/receiving the stream after the beginning of the time range do not receive the stream from its beginning. This kind of functionality is roughly equivalent to the one provided by a broadcaster, or by a TV service. Broadcast-like services can be divided in two sub-types:

- The producers upload a real media file to the server, and the server will start streaming it at the beginning of the time range (example: RTP/multicast);
- The producers do not upload a media file on the server, but they only upload a description of a live stream (an SDP, or a similar description). The producers will be in charge of streaming the media to the server (or directly to clients) from the beginning of the time range. This type of solution permits the producer to support real-time encoding. As an alternative, producers can use some P2P protocol for sharing the file directly with other pears, instead of streaming it to the server.

### 4.4 DAVS Architecture

In this section, we are going to present the overview of the DAVS system and describe the functionality of the different layers. We adopted the layered approach in designing the video server. Because the crux of the video server is to attain adaptability with different streaming engines with minimal effort, hence we can setup different experiments to measure and evaluate different parameters and to provision multimedia streaming services with QoS in mind in heterogeneous networks. Layered approach imparts us many benefits such as

- **Interoperability:**
  
  Layering methodology promotes greater interoperability between devices from different generations.

- **Compatibility:**

  One of the greatest of all of the benefits of using a layered approach in video
server design is the greater compatibility between devices, systems and networks that delivers the streaming services.

- **Flexibility:**
  Layering and the greater compatibility that it delivers goes a long way to improving the flexibility; particularly in terms of options and choices. Like later if we desire to add some modules in the DAVS API to acquire some statistics about the streaming servers, networks conditions etc. then the modification in the DAVS API will not require any alteration in any of two layers.

Now we described the functionalities of each layer that we opted to design the DAVS as exhibited in Figure 4.3 below.

![DAVS Layers Diagram](image)

**Figure 4.3: DAVS Layers**

### 4.4.1 DAVS API

To make the video server engine-agnostic, the video server interface must communicate with the engines through a well-specified interface: the DAVS API. The video server interface interacts with the DAVS API and the DAVS data base. The DAVS API only interacts with the available streaming engines on the server.

The encoder/transcoder interface used by the streaming engine will be based on **ffmpeg** [69], and is already fully specified; the video database keeps track the available streams in the streaming engine and their description.
The video server API is implemented by the streaming engine through a set of scripts which can be invoked by the server interface to feed the streaming engine, to activate the encoder or transcoder, to ask the system about available media streams, etc... The syntax and semantics of each one of these commands is fully specified below.

- **Validate**: This call permits to validate a media file or stream, identifying its format and syntax, and verifying if it is stream able (example: some streaming engines only support a limited set of audio/video codecs, some other engines require repeated global headers in the stream, etc...).

  *Command line parameters:* this call accepts an absolute path to the media stream (in the local file system) as an input.

  *Return value:* 0 if the stream is recognized and stream able, \(<0\) if the stream is not recognized (possibly corrupted stream), and 1 if the stream is recognized but not stream able (needs transcoding).

- **Import**: This call imports a media file or stream into the engine, making it available to clients.

  *Command line parameters:* this call accepts an absolute path to the media stream to be imported, and some optional switches:

  - the `-b` switch to specifies that the stream is a Broadcast-like stream;
  - the `-l` switch to identifies Broadcast-like live streams;
  - the `-t <br>` switch to force transcoding, and to specify the target bit rate `<br>`.
  - the `-g` switch to force the transcoder to transcode the video with the specified video codec `<video codec>`.
  - the `-a` switch to force the transcoder to transcode the video with the specified audio codec `<audio codec>`.

  *Return value:* 0 in case of success, \(<0\) in case of error.

  The input file is not modified nor deleted, the server interface is in charge of removing it; this call prints on standard output a stream ID that must be used to access this stream from other calls; for non broadcast like streams, this call prints on standard output the URI for accessing the stream (in the form `<protocol >://...`), or a description of the stream (an SDP description, or something similar).

- **Start**: This call, only available for broadcast-like streams, is used to notify the engine about the beginning of the time range in which the stream is active.

  *Command line parameters:* this call wants the stream ID of the affected stream as a parameter
Return value: 0 in case of success, <0 in case of error
This call prints on standard output the URI for accessing the stream (in the form 
<protocol >://...), or a description of the stream (an SDP description, or some-
thing similar)

- **Stop:** This call, only available for broadcast-like streams, is used to notify the 
  engine about the end of the time range in which the stream is active.
  Command line parameters: this call wants the stream ID of the affected stream 
as a parameter.
  Return value: 0 in case of success, <0 in case of error

- **Deport:** This call removes a media from the engine
  Command line parameters: this call wants the stream ID of the affected stream 
as a parameter
  Return value: 0 in case of success, <0 in case of error

To support a new streaming engine, it is sufficient to write the 5 scripts implementing 
validate, import, start, stop, and deport; the suggested names for the commands should 
be prefixed with a namespace that identifies the engine (example: live555_import, etc...).

### 4.4.2 Video server interface

Video server interface module is the focal entry point into a system. It has dual responsi-
bilities. One is to invoke the DAVS API based on the commands which are provoked by 
the DAVS client. DAVS client is a stand alone JAVA application which is designed and 
developed to test the DAVS server. Other is to communicate with the DAVS client and 
forwards the outcomes of the commands issued by the DAVS client, furthermore it is also 
responsible for the interaction with the DAVS data base. Next chapter will elaborate 
the whole procedure how the DAVS client interact with the video server interface and 
present the implementation challenges of the DAVS client application. The video server 
interface module and the DAVS client communicate with each other through well de-
defined protocol called remote procedure call (RPC). Next section will elaborate the whole 
process in detail how DAVS the client communicate with the video server interface and 
how they exchange messages with each other.

### 4.4.3 Streaming engines

This layer only incorporates with the available streaming engines in the video server. It is 
only responsible for streaming the media contents to the streaming client like in our case
is DAVS client, which is a test application to test the video server. Streaming engines start dispatching the media contents only when DAVS API invokes the streaming engine that streaming client desire to watch the following streams. All the available streams and their description are stored in the DAVS data base as shown in Figure 4.3 and presented to the DAVS client when it desire to view the available streams in the video server.

### 4.4.4 DAVS Database design

DAVS database consists of three tables named as DAVS Client, Video on Demand and Broadcast Streams. DAVS client table contains the information related the clients like user name and password, etc. who are authorized to access the video server. DAVS client table is introduced into the DAVS database because of the design and implementation of DAVS client application, which I described in detail in later section. Both Video on demand and Broadcast streams tables contain the StreamID filed, which is a primary key for both the tables. As stated previously StreamID is employed in the implementation of the DAVS API to uniquely identify the stream while the DAVS client request, to play a particular stream. The design of the DAVS database is shown in Figure 4.4 below.

DAVS video server interface is responsible for interacting with the DAVS database. The DAVS video server interface retrieved all the available streams and direct them to the DAVS client on request, an update all the tables when the DAVS client imports any video file or starts any broadcast session. As we are using MYSQL [75] database management system because MYSQL provides a C API, which is used to interact with the database [74].

As we can depict from Figure 4.3, messages exchanged between the video server and the DAVS client are through well known protocol called RPC [76]. The DAVS follows a layered approach and each layer is responsible for a specified task as mention in previous chapter. The DAVS video server interface encapsulates the responses from the DAVS API into a RPC message and delivers the RPC message back to the DAVS client.

There are two approaches for storing the multimedia contents into the database. One is to store the actual multimedia contents into the database, and other is to only store the path of the multimedia contents. First approach for storing actual multimedia contents into the database is not commonly deployed today. Because it leads to very large tables very quickly, that leads to all kinds of issues related to maintaince and portability. Hence, we opted to only stores the relative path of the media streams into the DAVS database.
As stated previously, DAVS video server interface also interacts with the DAVS database with a well defined API. During the communication between the DAVS client and the video server, whenever a client import any video to a server, or fetch all the available video on demand streams or broadcast streams, that lets the video server interface to issue a query using MYSQL API.

4.5 DAVS Client Design

The DAVS client is a pure Java application that is designed and implemented exclusively to interact with the DAVS server and to examine the distribution agnostic functionality. The DAVS client design not only allows us to fetch the all available streams including video on demand (VOD) and broadcast live streams on the DAVS server, although empower us to import different type of media streams to be distinct available streaming engines on the DAVS. All the classes that define for the DAVS client application according to their functionality are groups together under a package.

As stated previously that the DAVS client will communicate with the DAVS server through the video server interface. Hence, RPC protocol was employed between the video server interface and the DAVS client for message exchanging. The most challenging phase of the DAVS client implementation is to figure out how to implement ONC/RPC protocol and XDR routines in Java platform. As Java has its own technology called remote method invocation (RMI), to invoke remote methods/object from different hosts.
So to communicate with the video server interface, a third party tool is needed, which can propagate messages that is fully ONC/RPC protocol compliant. Luckily, I found an implementation of ONC/RPC protocol for the Java platform over the Internet called Remote Tea [81]. Remote Tea implements the complete ONC/RPC protocol for the TCP/IP and UDP/IP transports according to [76]. It not only provides the complete client and server side functionality but also has a \texttt{rpcgen} like a compiler called \texttt{jrpcgen}, which accept the .x file, which generate all the stub files of server and client side plus all the XDR routines in a form of classes. These all classes are grouped together in a package called \texttt{RPCClient}.

For security reasons, we design and implement the DAVS client application in such a way that only authorized user can access the DAVS server. At the beginning, user has to authenticate by providing the user name and password which is then matched with the stored ones in the DAVS database. If a user successfully authenticates himself, then determine the profile of the authenticated user. In the DAVS database, two profiles are defined, one is simple \textquote{User} and other is \textquote{Administrator}. After successful authentication, user profile is fetched from the DAVS database, which determines what rights should be given to that particular user according to that profile. If a user is authenticated, and he belongs to a \textquote{User} profile, then the DAVS Client only shows the available VOD streams and broadcast streams. Moreover he can also validates and import any media file into any available streaming engine. If an authenticated user is associated with the \textquote{Administrator} profile, then he can start or stop any broadcast like streams.

Design of graphical user interface (GUI) of the DAVS client application facilitates to fully utilize the DAVS server capabilities. The DAVS client designed with a variety of components, but some of the components are associated with a particular profile. After successful user authentication, thereafter applying profile to the authenticated user, this enables only those components which are authorized to use for that particular profile.

The DAVS client GUI has a drop down menu named as the \textquote{streaming engine}, which will be populated during the run time when user profile is determining for that particular authenticated user. The DAVS client invoked a method in \texttt{DAVS_\_Client.java} class called \texttt{GetStreamingEngines()}, which is defined in the \texttt{StreamingEngines.java} class. All the classes associated with fetching the information regarding the available streaming engines are defined in the \texttt{StreamingEngines} package.

Figure 4.5 shows a code snippet of retrieving the available streaming engine. \texttt{objDAVSEngines} invoke a method called \texttt{GetStreamingEngines()}, as the name suggests it would send a query to retrieve the available streaming engines on the DAVS server. The DAVS API has an additional script called \textquote{engines.sh}, which determine how many streaming engines are currently available on the DAVS server.
Chapter 4. DAVS Design

40

Figure 4.5: Available streaming engines on DAVS

The query initiated by the DAVS client at run time for the currently available streaming engines is a RPC request, which is directed to video server interface. Upon receiving the request, the video server interface (that is acting as an ONC/RPC server) invoke a script `engines.sh` that returns the name of the currently installed streaming engines on the DAVS server to the video server interface. After obtaining the names of the streaming engines from the DAVS API `engines.sh` script, the video server interface direct the response back to the DAVS client using ONC/RPC protocol. Figure 4.6 shows a code snippet of `engines.sh` bash script.

```bash
if [ ${#AvailableEngines[*]} -eq 1 ]
then
    echo "Error: No streaming engine available in DAVS server."
    exit 1;
else
    TotalEngines=${#AvailableEngines[*]}
    DAVS_Engines=""
    while [ $i -le $TotalEngines ];
        do
            engine=`basename ${AvailableEngines[$i]}`
            DAVS_Engines=${engine}$space$DAVS_Engines
            i=$[$i+1]
    done
    echo $DAVS_Engines
    exit 0;
fi
```

Figure 4.6: DAVS API script to retrieved the available streaming engines

If an authenticated user belongs to 'User' profile, he can only validate, import, view and play the available VOD media streams on the DAVS server, or he can join any active broadcast media session from the broadcast list menu on the DAVS client. If a
user shows his interest to validate any media file for any of available streaming engine, media file is initially copied to the /temp directory of the DAVS server using JSch [83].

JSch is a pure Java implementation of SSh2 protocol, which allows you to transfer any kind of file over the network to a remote machine securely. If a media file is validated for a particular streaming engine, then it can be imported to a particular streaming engine by simple copying that media file to a streaming directory, which contains all the media file that is available for the DAVS client.

If a user is interested to play any VOD streams or want to join any broadcast session, he first retrieved the stream URI of a particular stream that has been stored in the DAVS database during the import process. Stream URI is retrieved based on the stream ID from the database. Stream ID is generated for every media stream while importing into the DAVS database. The intention behind to generate a unique stream ID for every media stream is to help to distinguish every stream, in the future when huge number of videos will be imported into the server for either commercial deployment or experiment setup. Figure 4.7 shows snippet of code for generating stream ID in the DAVS API.

```
FileName=`basename $BroadcastStream`
ID=`head -c4 /dev/urandom | od -N4 -tu4 | sed 's/\[.*\]$/\n'
StreamID=${(FileName%*)}$ID
```

Figure 4.7: DAVS API import script for generating stream ID

To play a particular media stream, that user is interested to watch, a DAVSPlayer package contains a class name PlayStream.java, that invokes a Video Lan (vlc) [82] media player, a third party highly portable multimedia player for viewing multimedia contents and support of various streaming protocol. URI of a particular stream that is retrieved from the DAVS database is passed to the vlc player, which starts receiving the multimedia contents from the DAVS server.

If an authenticated user belongs to Administrator profile, then he can start or stop any broadcast session for any particular streaming engine by invoking <streaming_engine>_start.sh and <streaming_engine>_stop.sh respectively. These scripts are invoked through the video server interface. After initiating the broadcast multimedia session by the administrator other users can join this session by retrieving the URI from the DAVS database and passing this URI to PlayStream.java, which provoked the vlc player to join the broadcast multimedia session.
4.6 DAVS system overview

In this section, we exposed the whole DAVS system and illustrate in detail how the DAVS client acquaint about the current available streaming engines on the video server and how the available media streams are retrieved from the DAVS data base and presented to the DAVS client.

As we discussed earlier in the previous section, that the DAVS client does not interact directly with the data base. Rather the DAVS client issues the RPC messages to the video server interface module, which handles all the messages and sends the reply back to the DAVS client. In the same way, when the DAVS client application start it retrieves all the media streams that are available on the video server by issuing the RPC message to the video server interface, which fetches the available media stream from the DAVS data base, and presented to the DAVS client.

The DAVS data base only contains the information about the media streams like video file path (An absolute path of a video file), snapshot of a video file if and only if user desire to take the snapshot of a video file (An absolute path of a snapshot), and the particular streaming engine that will stream that video file. Figure 4.8 illustrate the whole DAVS systems.

- When the DAVS client application started, it invoked RPC call to fetch the available streaming engines from the video server. Receiving a RPC call from the client causes the video server interface to execute a script which inquire into the video server and examine the currently installed streaming engine on the video server and returns the response back to the video server interface which notifies the client by sending the RPC message back to the client.

- During the commencement of the DAVS client application, it forwards a RPC message to retrieve all the available media streams on the video server. Upon receiving this call, video server interface connect to the data base using MYSQL API, which retrieves all the media streams and sends the response back to the client through video server interface using RPC message.

- Client can now upload a video file to particular streaming engines that are available on the video server. Before uploading a video file, client is supposed to validate a video file for a particular streaming engine that he select from the available streaming engines on a video server. Now client invokes a RPC call, which causes a video server interface to call a ’<streaming engine>.validate.sh’ script. The intention of this script is to verify that a media file is stream able for a selected streaming engine.
After validating a media file client can import a media file into a particular streaming engine. Video server interface invoked a `<streaming engine>_import.sh` script, which simple copy a file to streaming directory of a particular streaming engine.

DAVS supports Video on demand (VOD) and broadcast/Live streaming. The DAVS client picks the stream either from the VOD streams or Live/Broadcast list which is populated as we mentioned in the previous steps. Next action will depend upon the stream type:

- **Video on Demand (VoD):** If the client is interested to see VOD streams then he can instantaneously start viewing the stream after pressing the 'Play' button on the client. And the specific streaming engine start dispatching the media contents to the DAVS client from the beginning.

- **Broadcast/Live Streams:** If client select the Live/Broadcast stream from the database, then he can start the stream by invoking the RPC call to induced video server interface to call `<streaming engine>_start.sh` script. The intention of this script is to start dispatching the media contents and generate a SDP file, which other clients will use to receive the media contents. This
script also updates the DAVS data base and the Live/Broadcast list on the client side so interested users can access the stream by playing the SDP file.

- If an administrator (if a client is login into a system with an administrator profile) is not interested in streaming the live video contents, then he can stop the stream at any time by sending the RPC message to video server interface which invoke the '<streaming_engine>_stop.sh' script, that will cause to stop dispatching the media contents of a particular streaming engines and also update the DAVS data base by removing the SDP file from the Live/Broadcast streams list.

- Administrator can remove the stream from the video server by invoking the '<streaming_engine>_deport.sh', which will update the DAVS data base and also remove the media file from the streaming directory.

Our proposed architecture eliminate the cost in terms of number of nodes that are configures with different streaming engines. Moreover, remove the cost of their maintenance. On the contrary, if we compare the design of DAVS with the video servers that are not "engine-agnostic", we can say that the DAVS design is so called framework and brings flexibility, because if we want to add some plug-ins/modules to get information about the multimedia contents into a video servers that are not engine-agnostic, we have to add /modules in each video server that all are configured with the same streaming technology. However, this is not the case with DAVS. We can attach different plug-ins into a video server that is supposed to support every hypothetical streaming software and technology to get data that are related to streaming experiments now new plug-ins or modules can be added easily.

4.7 Conclusion

In this chapter our proposed architecture was presented. We described the architectural components of the proposed system and presented the specification of the API introduced to manage the adaptability of the different streaming engines. In order to provide a better understanding of the DAVS functionality, block diagrams are also presented. Finally, a brief description with illustration of the whole DAVS system which includes the server and the DAVS client was presented. In the next chapter, we describe how we implement the whole system in detail and the challenges we confronted during the implementation.
Chapter 5

Implementation of Distribution
Agnostic Video Server (DAVS)

Design and architecture of the DAVS is described earlier. This chapter presents the implementation details and challenges faced during the implementation phase. First section discloses the details how the DAVS API is implemented, which described in detail in the previous chapter. Then the design and implementation details of the DAVS client discussed and how the DAVS client interact with the video server.

5.1 Implementation Approaches

In previous chapter, we present the whole DAVS design that is build by combining different modules that are well defined. In this section we will discuss the motivation behind our implementation approach for the DAVS.

The main intention behind this thesis work is to design and develop a video server, which can plug-in with different streaming servers and help us to accelerate to set up different streaming test beds for the experiment purpose. Previous chapter help us to elucidate the proposed design of the video server. As stated in previous chapter, the DAVS API is the crux of the whole proposed design. The DAVS API provides all the functionality to adapt different streaming engines.

In the previous chapter, the DAVS API functionality and the specifications of the programs in API is described in detail. The DAVS API composed of 5 programs. Each program has its own parameters and return values according to its functionality as fully specified previously.
From an implementation point of view of the DAVS API, there are many options available, but we opted to implement the DAVS API as shell scripts. It will provide comfort to many interested researchers to employ the DAVS API, because it does not require any special packages, unless the DAVS API just require a shell environment to get facilitated from it.

The DAVS API performed the video transcoding only if a video format is not complying with any of the available streaming engines. The DAVS API also supports the dispatching the live multimedia contents over the networks. All these features are employed in the DAVS API by the renowned program so called ffmpeg, which can convert and stream digital audio and video in various formats.

The most challenging phase of the whole project was to decide how the DAVS client communicates with the DAVS. One alternative was to design a protocol from the scratch that fully compliant with the DAVS design. Other approach was to get facilitated from the existing well defined protocol. We opted to go with the second approach as it would save the time for the design and implementation phase for the newly proposed protocol for the DAVS client. Hence, we introduced a RPC mechanism between the DAVS and DAVS client, because RPC is a straightforward mechanism to build distributed programs and impart efficiency in terms of enabling rapid communication.

Video server interface interacts with the both the DAVS data base and the DAVS client. As MYSQL connectors and API provides the connectivity to the MYSQL server for the client programs. Therefor we opted to design and build data base in MYSQL, because MYSQL API enable us to execute different statements from other programming languages.

Following sections will present the tools and environment that have been chosen and employed in the implementation of the DAVS and the DAVS client.

5.1.1 Shell Scripting

A shell is an environment in which we can run out commands, programs, and shell scripts. There are different flavors of shells, just as there are different flavors of operating systems. Each flavor of shell has its own set of recognized commands and functions. For example

\[\text{Korne shell} \quad /\text{bin/ksh or /usr/bin/ksh}\]
\[\text{Bourne again shell} \quad /\text{bin/bash or /usr/bin/bash}\]
bash is a shell: a command interpreter. The main purpose of the bash is to allow you to interact with the computer operating system or third part tools so that you can achieve whatever you are required to do. Shell script is a file, preceded by the pound sign #, and a list of commands, which are listed in the order of execution. When a shell script is executed, command interpreters go through the ASCII text line by line, loop by loop, test by test and execute each statement as each line is reached from the top to bottom [72].

5.1.2 FFmpeg

FFmpeg is a whole package, and cross-platform solution that impart us to record, convert and stream audio and video. It includes libavcodec- the leading audio/video codec library [73]. FFmpeg project was initiated by Fabrice Bellard, and is now maintained by Micheal Nierermayer. FFmpeg project comes from the MPEG video standards group, where FF stands for “fast forward”. FFmpeg is developed under GNU/Linux, but it can be compiled under most operating systems, including Apple Inc, Mac OS X. Microsoft Windows and Amiga OS [68].

FFmpeg project is composed of several components; like ffserver is as HTTP and RTSP multimedia streaming server for live broadcasts [70]. ffplay is a simple media player based on Simple DirectMedia Layer (SDL) and on the FFmpeg libraries [71].

FFmpeg is a very fast video and audio converter. It can also capture from a live audio/video source. FFmpeg command line interface is designed to be visceral, in the sense that FFmpeg tries to compute all parameters that can possibly be deduced automatically. FFmpeg determines all other parameters by itself, you usually only have to specify the target bit rate you want. FFmpeg can also convert from any sample rate to any other, and resize video on the fly. FFmpeg supports a wide variety of file format and protocol that can be specified as an input in the command line interface [69].

5.2 API Implementation

The design of the DAVS is built upon layered structured. Each layer provides a set of functionalities as described in previous chapter. The DAVS API consists of five scripts, which are written in Bash scripting language. According to the specification of the DAVS API, that does not require any extensive programming. Our major work is to do with video trans coding which we employed in the DAVS API by taking advantage of the third party tool so called FFmpeg, which impart us a great flexibility to trans code to be possible to any format.
As we know bash scripting provides a great flexibility to the end user, and you can achieve your desired results very quickly in bash scripting as compared to other languages. You can easily interact with third party tools within a script. So these features urge me to use bash scripting.

The design and implementation of the DAVS is done under the GPL license [66]. The basic intention behind to make the whole implementation details open source and publicly available is to let others to define and implement different plugins and modules for the video server. Appendix A provides the details of the public repository from where any one can downloads the source code of both the DAVS and DAVS client.

Now I will describe in detail how I use the bash scripting language to implement the DAVS API to achieve engine agnostic capability that is to easily plug in order to different streaming engine.

5.2.1 Validation

As stated earlier, validate.sh scripts should be affixed with the namespace that determines for which streaming engine DAVS API is currently engaged with. `<streaming _engine >_validate.sh` accepts four parameters as an input with the well defined switches as illustrated below.

```
Usage: validate.sh [-v Enable verbose mode. ] [-f /path/of/a/file/to/validate ] [-s Take snapshot of a media stream. ] [-h for Help]
   -v Enable the verbose mode.
   -f Specify the path of a file to validate.
   -s Take snapshot of a video if file is validated.
   -h Help.
```

This script accepts a file to be validated for a specific streaming engine as a parameter with `-f` switch; a file can be any media file or SDP file. `<streaming _engine >_validate.sh` first determines that whether the file specifies in the input is a media file or SDP file. The whole validation process depends upon the file type. If a specified file is a SDP then scripts verified that format of the SDP file and the parameters in the file are according to the RFC 2327. If the parameters and the format of the file are not according to the standard then it will print an error message on the console and report to video server interface, if and only if the verbose mode is ON. Otherwise, script will return with exit status `<0`, which states that SDP file, is corrupt or does not meet the standard. Besides, if all the parameters are specified in the file meet the standard then it returns with exit status `0` and informs the video server interface, which send an
appropriate message back to the DAVS client, that states that current streaming engine for which validation process is taking place can understand this media file.

If the file is a file then validation process is slightly different and complex than validation of SDP file. In the case of media file script first read the config file, which contains the information regarding which audio and video codec are supported by the streaming engine. In order to validate the media file that current media file is streamable with the streaming engine, I used a third party tool FFmpeg in the script to retrieve the audio and video codec information of the media file. After fetching the codec related information from a media file we checked that whether the retrieved information matches the audio and video codec specified in the config file. If it matches then we take the snapshot of the media file and stores in a directory that is mention in the config file, if and only if –s switch is specified, and report to video server interface with exit status 0. If verbose mode is ON then an appropriate message is also printed on the console before informing the video server interface. A complete flow of the script is shown in Figure 5.1 below.

5.2.2 Importing

Afterwards <streaming_engine>import.sh scripts will be invoked to import the media file into the streaming engine. You can specify whether the media file you are interested to import into the streaming engine is whether a broadcast stream or video on demand file. import.sh implement in such a way that it provides flexibility, because it has many
switches which accepts different parameters for trans coding for either broadcast media stream or video on demand stream. Different parameters that can be passed with different switches and their description are shown below.

Usage: import.sh [ -l Live Streaming ] [ -b Broadcast streaming ] [ -d Video on Demand ] [ -t Bit Rate ] [ -g Video codec. ] [ -a Audio codec. ] [ -v Verbose mode on ] [ -h help ]

-  
  Broadcast like Live streaming.
-  
  Broadcast streaming.
-  
  Video on demand (VOD).
-  
  Set the bit rate in Kbits/sec, for example import.sh t 200
-  
  Set the video codec, for example import.sh g mpeg1video
-  
  Set the audio codec, for example import.sh -a mp3
-  
  Enable verbose mode.
-  
  for Help.

When `<streaming _engine >_import.sh` script is invoked with the `-d` switch, which indicates that media stream is a video on demand stream. We can pass different parameters, which are required for trans coding with different switches to help the media stream to be stream able to the current streaming engine. This trans coding process will only occur if the validation.sh script informs the DAVS server interface that media stream need trans coding by sending the return code 1. If media stream needs trans coding and parameters are not specified while invoking the `<streaming _engine >_import.sh`, default parameters will be used, which are specified in the `config` file.

As stated in previous chapter that broadcast stream can be classified into two types; one is where only a description file like SDP, etc. is uploaded into the server and other is when actual media stream is uploaded into the server. To accommodate these two types of streaming types, two more switches are implemented in `<streaming _engine >_import.sh` script, where you can specify the media type either it is live broadcast (like SDP file) or simple broadcast where actual media stream is passed to the script.

 `<streaming _engine >_import.sh` script accept description file like SDP with the `-l` switch. If `<streaming _engine >_validate.sh` script return with the exit status 0, which indicates that session description file follows the standard, then `<streaming _engine >_import.sh` copies the session description file to the streaming directory which is mention in the `config` file. And inform the DAVS server interface that file is available for the DAVS client.

If `-b` switch is applied before commencing the `<streaming _engine >_import.sh` script, which indicates that an actual media stream is going to be imported into the server. Before copying this media stream and made available for the DAVS client, I
make sure that this media stream is streamable with the streaming engine by examining the exit status of <streaming_engine>_validate.sh script. If this media stream needs some trans-coding then different parameters can be passed to the script with available switches as mentioned above. If media stream needs transcoding and no parameters are specified then the default parameters are being retrieved from the config file. After trans-coding the media stream script inform the DAVS server interface by return the exit status 0, that media stream is now available for the server. A unique ID is generated for each stream when media stream is importing into the server. Later stream id can be used to identify a particular stream. A complete flow of the import program is shown in Figure 5.2 below.

*Figure 5.2: Flow of DAVS API Import script*

### 5.2.3 Start

<streaming_engine>_start.sh scripts will invoke only for the broadcast type streaming. Streaming Id is passed to the script which induced the streaming engine to start a media session within the specified time range as mention in the broadcast session. Different parameters that can be passed with different switches and their description are shown below.

*Usage: start.sh [-s Stream ID] [-v Enable verbose mode.] [-h help]*

- `-s` Notify the streaming engine to start the stream.
- `-v` Enable verbose mode.
- `-d` for Help.
As a result URL is generated by the script for accessing the media session and informs the DAVS video server interface that media session has been started and the DAVS client can join the session by this URL. A complete flow of the start script is shown in Figure 5.3 below.

\[Figure 5.3: \text{Flow of a start script of DAVS API}\]

### 5.2.4 Stop

As the name suggest that `<streaming_engine>_stop.sh` script instructs the streaming engine to stop the broadcast session. `<streaming_engine>_stop.sh` accepts a streaming id as an argument, which is a unique ID generated during the importing process for each stream.

### 5.2.5 Deport

`<streaming_engine>_deport.sh` script is used to remove the stream from the video server. After removing the stream from the video server it will inform the DAVS video server interface that stream has been removed with exit status 0, which will cause to update the DAVS client using RPC message. The DAVS client implementation and how the DAVS server interface communicate with it will be discussed in later section. Flow of the deport script is shown in Figure 5.4 below.
5.3 Video Server Interface

Communication process between the DAVS and client is built based on RPC protocol, which is defined by in [77]. RPC is a compelling technique and popular paradigm for designing and constructing distributed client-server based applications. RPC extends the concept of a conventional, local procedure calling mechanism.

Implementation of RPC does not require that procedure need to be in the same address space as the calling procedure. The two processes may be on the same system, or they may be on different systems with a network connection between them. Designing and Implementing of a client server model which entails more computing; remote procedure call (RPC) imparts many advantages to such a model by giving straightforward semantics make it easier to build and maintain correct distributed programs and efficiency is achieved by making procedure call mechanism appear simple enough to enable rapid communication.

Remote procedure call (RPC) is alike to a conventional functional call. When RPC is made, the calling arguments are passed to the remote procedure and the caller waits for a response to be returned from the remote procedure. Client makes a procedure call that sends a request to the server and waits for a response from a server. When the server accepts the request, it invokes a routine that performs the requested service and
Figure 5.5: Basics behind RPC client server program [78]

sends the reply back to the client. Figure 5.5 depicts the whole procedure that takes place when RPC call between two networked systems.

Figure 5.6: Remote procedure call (RPC) mechanism [79]

Figure 5.6 illustrates the basic operation of RPC. A client application issues a normal procedure call to a client stub. When client application calls a procedure, client stub receives arguments from the calling procedure and converts the input parameters from the local data representation to a common data representation. This whole procedure is
defined as *marshalling*. Networking functions in the O/S kernel is invoked by the stub to dispatch the message. Kernel directs the message to the remote server. After receiving the messages network routines on the server invoke the server stub which un-marshals the parameters from the network message and execute a local procedure call. When the procedure completes its functionality, it returns its execution to the server stub and server stub marshals the return values from the procedure into a network message. The return message is sent back to the client. After receiving the network message from the server, client stub un-marshaled the parameters. The return values from the remote procedure are set on the local stack.

RPC is an independent of transport protocol; that is RPC does not concern about how a message is passed from one process to another or on a machine to another machine; RPC is pertained only with the specification and interpretation of messages.

A remote procedure is uniquely identified by the triple: (program number, version number, procedure number). An integer identifier is attached to a RPC program that is known to the programs which will call its procedure. Each procedure is also assigned a number that is also known by its caller. ONC RPC\(^1\) uses a program called *portmap* to allocate port numbers for RPC programs. When an RPC program is started, it registers itself with the *portmap* process running on the same host. The portmap process then assigns the TCP and/or UDP port numbers to be used by that application.

XDR empower RPC to handle arbitrary data structures, regardless of machines byte order or structure layout conventions, by converting the data structures to XDR before sending them over the network. XDR is standardized in RFC 1014 [80] for the description and encoding of data. And the process of converting from a particular machine representation to XDR format is called serializing, and the reverse process is called de-serializing.

The most challenging phase of building a RPC application is to write XDR routines that convert procedure arguments and results into their network format and vice versa.

*rpcgen* is a compiler that helps to automate the process of writing RPC applications. With *rpcgen*, the compiler does most of the dirty work; the programmer just needs to focus on debugging the main features of the application. *rpcgen* accepts a remote program interface definitions written in a RPC language, which is similar to *C* and produces C language output for RPC programs. The output file generated by the *rpcgen* compiler consist of a stub version of the client routines, a server skeleton, XDR routines for parameters and results, a header file that contains common definitions, and *ANSI C* prototypes stub routines.

\(^1\)ONC RPC is a well known implementation of RPC on UNIX systems. It was first initially called as Suns RPC and used as the basis for Suns NFS.
rpcgen accepts a special file with an .x suffix acts as a remote procedure specification file. Figure 5.7 shows some of the contents from the DAVS.x file in which declaration of functions according to the RPC language specification. As stated in previous chapter that video server interface also interact with the DAVS database so all database connectivity routines are also declared in this file. After declaring all the routines that are needed to interact with the DAVS API and the DAVS database, rpcgen is employed to DAVS.x, which generates all the XDR routines, server stub, etc. Appendix A provides the details of the public repository, where any one can access the complete specification of DAVS.x file and implementation of the Video server interface.

As generated files of rpcgen compiler contains both the server and client side stub. Video server interface as stated previously in the DAVS design received RPC messages from the DAVS client application and interact with the DAVS API and database. Therefore, the video server interface is implemented as a RPC server which received a request from the DAVS client. During the implementation phase main intention was to implement the server stub davs_server.c, which performs the marshalling and un-marshalling of parameters passed and the result during the interaction between the DAVS API and specially the DAVS database.

When the DAVS client (which acting as a RPC client, more detail will be discussed in later section) validate any video file for a particular streaming engine, it sends a request to the DAVS. Upon receiving the DAVS client request, video server interface, which is acting as a RPC server, a DAVS server stub davs_server.c unmarshalls the parameters
from the network message and invoke a particular procedure that is identified by triple value (which includes a program number, version, and procedure number).

Video server interface implementation is purely compliant with ONC RPC, which is Sun Microsystems initiative for their Network File System project. It is based on calling conventions used in UNIX and the C programming language. As the DAVS API is collection of bash scripts, which brings the distribution agnostic capability into a video server. Since C provides a variety of functions to run another program in C, so in order to invoke bash scripts, a function called system() is employed in video server interface as shown in Figure 5.8 below, all the parameters that are sent by the DAVS client are passed to the DAVS API script.

```c
//If user is interested to take snapshot of media stream.
if ((TakeSnapshot==1)) {
   srcut(DAVS.Validate_Script,TakeSnapshot);
}

BashExit��system(DAVS.Validate_Script);
wait(&BashExit);
```

**Figure 5.8:** Video server interface invoking DAVS API *validate.sh*

system() function does not allow to return more than one value. As some of the API scripts return only the integer value indicating that perform action is successful or not. However, some API scripts also return the URI (Uniform Resource Identifier) for a particular stream either importing video stream into the DAVS or while commencing a live broadcast session. Consequently, defined structure `ScriptResult` in `DAVS.x` helped me to retrieve the exit status and the URI of the DAVS API script. Later server stub does the marshalling processing to inform the DAVS client that media stream is now available for the video server, and it can be accessed through this URI.

```c
static ScriptResult result;
FILE *fp;
fp=fopen(DAVS_Import_Script,"r");

while(!fgets(ScriptOutPut,sizeof(ScriptOutPut),fp)) {
   strcpy(result.CommandOutPut,ScriptOutPut);
}
result.ExitStatus=pclose(fp);
return &result;
```

**Figure 5.9:** Video server interface invoking DAVS API *import.sh*
Figure 5.9, which shows a code snippet of video server interface implementation, shows how `popen()` function is utilized to store the exit status and the output of the DAVS API import script. First a FILE pointer is created to store the output of the DAVS API script. `popen()` function accepts the name of the script as an argument. All the parameters that are needed for that particular script are concatenated with the script. After executing the script, `popen()` function returns the FILE pointer, which holds the information about the script output.

However to retrieve the output of a particular script, `fgets()` function helps us to fetch the message from the FILE pointer. It accepts the character array, size of the character array, and the file pointer `fp`. Exit status of the script is retrieved by closing the file pointer. Script message and the exit status of the script are then stored in the structure `ScripResult`, which is defined for this purpose in `DAVS.x`. Thereafter filled structure will be directed to the DAVS client by performing the marshalling process. Later the DAVS client receives the message, and determines the exit status of the script and show the message accordingly. Next section presents the implementation details of the DAVS client.

### 5.4 Proposed packages of DAVS Client

As mentioned earlier DAVS consists of a number of packages and each package contains a number of classes that have been designed and implemented for a specific reason. Figure 5.10 shows all the packages and description of each package is under as follows,

- **DAVSClient:** This package contains the classes related to the GUI of the DAVS client. `DAVS_Client.java` is not only the actual starting point of an application but also all the logic and the decision that has been made during the session with the DAVS server is also implemented in this class.

- **DAVSMediaCopy:** This package includes all the classes related to securely transfer multimedia file to the DAVS server. `SecureMediaTransfer.java` class uses `JSch` package for this purpose.

- **RPCClient:** This package includes all the files that are generated by the `jrpcgen` compiler for communicating with the video server interface.

- **DAVSLogin:** It is responsible for authenticating the client and retrieving the profile of authenticated user from the DAVS database.

---

1. `popen()` function executes the command specified by the string command. It creates a pipe between the calling program and the executed command, and returns a pointer to a stream that can be used to either read from or write to the pipe.
• **DAVSPlayer**: This package contains only one class `PlayStreams.java` that is only responsible for invoking third-party multimedia player called `vlc` for viewing VOD streams or broadcast sessions.

• **DAVSDatabaseServer**: It includes a class that dispatches the request to video server interface using `RPCClient` package. As I stated previously, video server interface is responsible for interacting with the DAVS database.

• **StreaminEngine**: This package includes all the classes that not only interact with `RPCClient` package that initiate the request for particular the DAVS API script for retrieving the available streaming engines on the DAVS server.

Appendix B shows the class dependency diagram that help to understand the relationship among classes with in application and snapshot of DAVS client application.

![DAVS Client Packages](image)

**Figure 5.10**: DAVS Client packages
Chapter 6

Testing of Distribution Agnostic Video Server (DAVS)

A primary concern for any video server is to measure the capacity with regard to the quality of the service specification before it is being deployed to impart multimedia services to its intended users. This chapter presents the testing and evaluation of DAVS, with special attention to limitation induced by plug in different streaming engines into a video server. The evaluation is based on a set of metrics designed to determine if DAVS architecture introduced any overhead in hardware utilization of a server. To validate the effectiveness of these metrics, a prototype of multimedia streaming services has been built with the DAVS and the load for DAVS has been characterized. Our results show that DAVS architecture did not introduce any overhead on the hardware resources, when it was employed with different streaming engines in a live test bed.

6.1 DAVS Performance Evaluation

For the successful deployment of any video server, we need to measure the capacity of video server before it is being deployed to launch multimedia streaming services. The two main factors that determine the capacity of a video server; number of client’s requests the video server can serve, and the quality of the service perceived by the client while provisioning streaming services by the server. The greater the number client’s requests handled by the server and the higher the quality perceived by the clients, would be consider as an economical solution.

The staple motivations behind this research work are to

- Design and implement an API that can be plug-in with different streaming engines.
• Design an architecture of a video server that how can this API will be integrated with the video server called Distribution Agnostic Video Server (DAVS).

• Test and evaluate the performance of the DAVS.

DAVS will be employed in different multimedia streaming experiments in a large scale with different streaming engines in future. Therefore, we need to prefer those salient aspects of a video server that ensure the availability of a video server, when large numbers of the DAVS clients interact with different streaming engines.

One of the goals of prototyping was to evaluate the performance characteristics of a video server. To the best of my knowledge our work is the first step in the development and testing of distribution agnostic video server (DVAS).

We are primarily concerned to test the "engine agnostic” functionality of the video server and measure the performance with respect to the utilization of hardware resources of the DAVS. Also, our proposed architecture of video server will be deployed in future in a number of streaming experiments with the large number of multimedia streaming contents. Hence, it is essential to evaluate the video server on whether hardware resources get overwhelmed by the DAVS client’s request, in the situation where the video server is configured with multiple streaming engines.

We would also like to determine the behavior of the hardware resources utilization of the DAVS, while provisioning multimedia services by employing different streaming engines. To achieve this objective, we measure and then analyze the performance metrics against the following experiment.

6.1.1 Metrics of DAVS capacity

The aim of these metrics is to measure the load of the DAVS with an increased number of DAVS client requests. By monitoring the DAVS, the load of its resources (CPU, network, memory) and the utilization of each of them can be evaluated.

CPU utilization: This is the percentage of DAVS processor utilization.

Network utilization: This is the percentage of network utilization measured in the DAVS network interface.

Memory utilization: This is the percentage of DAVS memory utilization.
6.2 Experimental Procedure

Experiment was set up in a live network of Telecommunication System Laboratory (TSLab), KTH, Royal Institute of Technology. The experiment was carried out with one machine which was configured as a DAVS and other 50 machines were acting as DAVS clients. Figure 6.2 illustrates the placement of a video server in the network.

In the experiment, DAVS client can play as much streams as he wants or can join any active broadcast multimedia session. The choice of multimedia stream is made randomly among all available multimedia streams. During the experiment we seen that, the access frequency of any multimedia stream is proportional to the popularity and the more recent the stream, the greater its popularity. In Figure 6.1, DAVS client interaction is shown with the DAVS. The diagram shows the possible states of a DAVS client and the interactions of the DAVS client with the DAVS that makes the client state change.

![DAVS client interaction diagram](image)

During the experiment, measurements were taken from the DAVS using the third party tool **dstat**\(^2\), **sar**\(^3\) system utility and **ifstat**\(^4\).

---

\(^1\)TSLab - a creative laboratory, KTH, Royal Institute of Technology; http://www.tslab.ssvl.kth.se

\(^2\)dstat: Versatile resource statistics tool; http://dag.wieers.com/home-made/dstat/

\(^3\)system activity report; http://linux.die.net/man/1/sar

\(^4\)ifstat(1) - Linux man page; http://linux.die.net/man/1/ifstat
Chapter 6. Testing of DAVS

- **dstat**: instantly gives you the detail information about the consumption of all the system resources and display the information in different columns which clearly indicates about what magnitude and unit the output is displayed.

- **sar**: can monitor several system functions related to overall system performance.

- **ifstat**: is a tool to report network interfaces bandwidth.

![DAVS experiment setup diagram](image)

Figure 6.2: DAVS experiment setup diagram

### 6.2.1 Experiment Description

We deployed the video server in the live network as we described in previous section. For the successful experiment, we employed 50 users in our DAVS system with different profiles, which will test our video server. After creating the users, we emailed the URL to these clients where they can download the DAVS client software which is specially design to interact with the video server, for doing testing related activities like video upload, download, streaming, etc. Initially we put some raw videos for streaming. We are maintaining the user’s profiles on our DAVS database system and each user has its login and password to perform video manipulation on DAVS.

Our video server was implemented on a Intel(R) Core 2 T5500 CPU 1.66GHz with 2GB of RAM. Test bed network is connected by a Broadcom NetXtreme Gigabit Ethernet link. In this experiment, we deployed two streaming engines; one is LIVE555 Media
Server\textsuperscript{[84]} and other is RoSE\textsuperscript{5}, which is University of Trento proprietary streaming engine, which not only supports video on demand (VOD) but also supports the live broadcast streams. The objective of this experiment is to determined the behaviour of hardware resources utilization of the DAVS. The specification of a video server is shown in Table 6.1 below.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Intel(R) Core 2 T5500 CPU 1.66GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>2 GB</td>
</tr>
<tr>
<td>Cache size</td>
<td>512 KB</td>
</tr>
<tr>
<td>Operating System</td>
<td>Ubuntu 9.10</td>
</tr>
</tbody>
</table>

Table 6.1: DAVS test bed configuration

6.3 Experiment Results

Results have been obtained from a preliminary analysis of the experiment. Figures have been drawn that shows the values of some metrics as a function of the number of users simultaneously connected to the DAVS.

During the experiment, we have employed the video server with two different streaming engines that not only facilitate the video on demand (VOD) requests, but also capable of dispatching live multimedia contents. To measure the capacity and scalability of the video server, we were continually monitoring the behavior of the hardware resources of the video server; whiles DAVS clients were interacting with the video server.

Only 50 users were participating in the experiment, but the DAVS database expanded in no time. We had seen that participating users were not only interested in video on demand streams but also uploading the multimedia contents into the available streaming engines. We were constantly monitoring the behaviour of the DAVS database and the response time of retrieving the available VOD streams and active live broadcast multimedia streams from the DAVS database and its impact on the utilization of hardware resources of the DAVS, while users were uploading their multimedia contents into the available streaming engines. However, DAVS client requests did not overwhelm the hardware resources of the DAVS. Because we did not store the multimedia contents into a database, rather we opted to store the multimedia stream path in the DAVS database.

\textsuperscript{5}RTSP server, 
\texttt{http://imedia.disi.unitn.it/rtsp-v0.3.tgz}
6.3.1 Behavior of CPU and Memory utilization

In this section, we analyze the information that we acquired during the experiment. We discussed the behavior of the hardware resources (CPU, memory), when DAVS was deployed in a live test bed that not only supports VOD requests but also facilitate live multimedia contents to the participation users.

The experiment was carried out by the 50 users, and we were monitoring the behavior of the hardware resources of the DAVS, while DAVS clients were increasing by the time. Gathered data is plotted on the graph as shown in Figure 6.3 and 6.4, which displayed the utilization of the CPU resources and memory utilization.

In Figure 6.3, total number of participating users has been placed on X-axis. The impact on CPU resources and memory utilization by the increase of DAVS clients has been drawn on Y-axis. During the experiment, users were not also viewing available video on demand (VOD) streams, but also active live multimedia content. Rather many of them were uploading their contents into the DAVS. As DAVS API is the crux of the architecture of the DAVS, which not only invoke the streaming engine to stream the multimedia contents but also validate the contents whether its streamable with the current streaming engine. If any video is not streamable with the current streaming engine then DAVS API performs the transcoding, which requires some resources to perform transcoding.

After analyzing Figure 6.3, we discover that there was a sudden drop in CPU resources of the DAVS in variable times during the experiment. The abrupt change in the CPU resources is due to those DAVS clients’ requests, which requires DAVS API to transcode the video content to make it stream able for the selected streaming engine. Since the algorithms of transcoding have become more efficient, they have also become more complex which demands more processing and memory resources.

During the experiment, we observe that the video server did not experience the saturation. However, we can see from Figure 6.3, there is an abrupt fall in the availability of CPU resources, when number of DAVS clients increased. Nevertheless, the utilization of CPU resources did not fall below 90%.

The sudden fall in the CPU resources is due to the transcoding, which demands more processing and memory resources. Transcoding process is initiated by the DAVS client, when the multimedia stream is not streamable over the current available streaming engines. However, when DAVS clients were viewing the available multimedia streaming contents, then most of the CPU resources were in idle state.
When DAVS clients increased during the experiment, the behavior of the utilization of the DAVS memory is shown in Figure 6.4. The gathered information depicts the relationship between the memory utilization and the simultaneously connected users with the DAVS. As we can see in Figure 6.4, the association between memory utilization and connected users forms a linear relationship between them. The linear association express that when a larger number of clients connected with DAVS, more memory resources were needed.

Transcoding process requires the decoded data to be stored in memory. From Figure 6.4, we see that requirement of memory increased with number of DAVS clients requests, when they were importing multimedia stream into available streaming engines that requires transcoding. DAVS API transcode the multimedia streams which induced more memory consumption. Moreover, when different available streaming engines starts dispatching streaming contents which also requires some memory resources. However, it does not fully utilize the memory resources in the experiment. Nevertheless, one should interest to determine the number of users that exhaust the memory resources of the DAVS.

It can be seen from Figure 6.3, that our proposed design and implementation of the DAVS did not saturate the CPU resources of the video server. During the experiment, when the video server were dispatching multimedia contents to the DAVS clients or clients were uploading their streaming contents into the available streaming engines, the basic hardware resources including (CPU, memory, as shown in Figure 6.4) did not get saturated.

The results lead us to conclude that DAVS can be deployed in large scale experiments with different streaming engines, and this would not impose any overhead on the hardware.
Figure 6.4: DAVS memory utilization

resources of the video server.

6.3.2 Behavior of DAVS Network Interface

Moreover, we also examined the behavior of the network interface of the DAVS. We are interested to determine whether a video server that is configured with different streaming engines would not exhaust the attached network interface card.

During the experiment, we also populated the DAVS database with some high definition video on demand (VOD) streams. To determine whether network interface card is exhausted by the DAVS clients requests for these high definition video streams. However, fortunately DAVS’s attached network interface card is not overloaded by the DAVS clients requests. Which not only include importing the video streams into available streaming engines but also dispatching multimedia contents to the DAVS clients by the DAVS.

Figure 6.5 exhibits the network interface utilization in percentage. On X-axis, we draw the number of users that are connected with the DAVS and its impact on the attached network interface card on Y-axis.

Many factors can determined the behavior of network interface card utilization, like how many streams are currently streaming by the streaming engine at that moment, DAVS clients requests, and bit rate of each available streams on the video server. During the experiment, DAVS database has large number of multimedia streams and each available stream was encoded with different bit rates that were employed to determine the network interface card utilization. Different DAVS clients requests were initiated
to stream the contents from the video server and to examine the network interface card utilization behavior.

As we can see in Figure 6.5, it behaves linearly until reaching the stabilization point. We have concluded from the results that relationship between the user’s interest in available streams and network interface utilization forms a linear association. It entails that network interface utilization will increase with the increase in DAVS clients. However, we think one should interest in the maximum number of multimedia session can be supported and can be one of the possible promising future works.

![Figure 6.5: DAVS network interface utilization](image)

The utilization levels of DAVS resources remain approximately stationary beyond the stabilization point, but none of them show a utilization level near saturation. We have just seen the results in terms of the number of streams and utilization of hardware resources on our designed DAVS. Video contents were transferred from the server to 50 clients each of which resides on a separate network and performance metric is measured.

### 6.4 Advantages and disadvantages of DAVS

The following are the advantages of DAVS architecture:

- Single video server that can plugin different streaming engines.
- Decrease in the cost of video servers.
- Low maintenance and management cost.
The following are the disadvantages of DAVS architecture:

- Single point of failure.
- Current implementation of the DAVS API only supports Linux and Unix platforms.

6.5 Conclusion

A video server must provide real-time delivery of video streams through the interconnecting network and to the clients display. The implemented video server in this thesis work will provide an exactly end-to-end performance guarantee. It can be concluded that utilization of hardware resources of the DAVS does not reach saturation level, even when DAVS was tested with multiple streaming engines.

The major contribution of this thesis work is the design and implementation of DAVS and making the source code available publicly under the GLP license. In particular we present the results of the performance metric on the defined test bed.

In general, we are encouraged by these results and believe that further work needed to evaluate the video server performance with different performance metrics.
Chapter 7

Conclusion and Future Work

In this chapter, we are going to present the conclusion of our work, and then we will talk about the areas that can be enhanced in the future work.

7.1 Conclusion

The objective of this thesis work was to design and implement a video server that can dynamically couple to different streaming engines to conduct different streaming experiments. In this thesis work, modular approach was opted, which really helped us to bring an engine agnostic support into a video server. The adopted approach really helps us in the implementation phase, because all the modules are well defined. We also implement a client application that is purely designed to interact with DAVS. As part of the work, we also deployed the video server in a live test bed for evaluating our design and test the "engine agnostic" functionality of the DAVS. The results which we obtained from the test bed did not pose any load on the video server. We believed that our proposed video server not only being employed by the research community for the different experiment purpose. Different enterprise can take benefit from our video server as well.

Today we have witnessed that almost in all types of businesses, multimedia streaming applications are making an impact in their business model. This new concept will take the multimedia streaming applications into a new horizon, because this video server will not confine to only one media format and streaming technology. As a result, multimedia streaming services can be delivered not only over the Internet (i.e., video on demand, broadcast TV) but also in the enterprise (i.e., marketing of a campaign, training, event broadcasting, product information, video conferencing, etc.) that will be independent of distribution technique.
7.2 Future Work

Probing deeper, the results in this thesis also provide a strong foundation for future work in streaming server. This thesis work provides a framework to unfold many opportunities in a new paradigm of video streaming server. This is just a beginning so a lot of work has to be done to extend the current implementation. Some suggestions are:

- Today peer to peer streaming is dominating over the Internet. A new plug-in or module can be designed and implemented is such a way that peer to peer streaming engine will be invoked when number of hits of a popular media stream reach at some threshold. Each DAVS client will be acting as peer to peer client in order to retain the availability and stability.

- DAVS can also be deployed in different test bed for examining the different streaming characteristics in different access networks. So new modules can be added in the DAVS API to extract different parameters like latency, delay, etc. while DAVS dispatching multimedia contents to the end user.

- As we have tested the DAVS with a limited number of users in a test bed, so another future improvement will be to determine the bottle neck of a server. Because it is very important as the video server is not bounded to one streaming engine and media format.

- HTTP interface should be added in the video server interface in order to access the streaming contents through the web.
Appendix A

DAVS Code

All the source code of the DAVS is now publicly available under the GPL license version 2. We have created a project on the github code and uploaded all the source code on it with the consent of Renato Lo Cigno and Luca Abeni in University of Trento, Italy\(^1\). Now every one can access the code and make changes under the GPL license. To access all the details of the DAVS project page please visit DAVS\(^2\).

\(^1\)http://disi.unitn.it/
\(^2\)Distribution Agnostic Video Server (DAVS)
http://github.com/
Appendix B

DAVS Client

B.1 Class dependency Diagram

![Class Dependency Diagram]

Figure B.1: DAVS client Class Dependency Diagram

B.2 Snapshot of DAVS Client
Figure B.2: DAVS client snapshot
Bibliography


[16] Internet Media-on-Demand; The Real Time Streaming Protocol; URL: http://www1.cs.columbia.edu/~hgs/teaching/ais/slides/2003/RTSP.pdf; Last access: June 2009


[20] Colin Perkins; "RTP: Audio and Video for the Internet"; Addison Wesley; June 12, 2003

[21] H. Schulzrinne, S. Casner; "RTP Profile for Audio and Video Conferences with Minimal Control: RFC 3551"; July 2003;


[27] Multicast Routing In a Datagram Internetwork; PhD. Thesis. Stanford University. 1991


[34] The Explosive growth of streaming media; URL: http://digitalcontentproducer.com/mag/video_explosive_growth_streaming/; December 2009


[36] Napster LLC. Napster Free; URL: http://free.napster.com/

[37] Bram Cohen. BitTorrent; URL: http://www.bittorrent.org

[38] Alexender Engman; ”High Performance Video Streaming on Low End Systems”; Master Thesis Electrical Engineering Department, KTH Royal Institute of Technology; December 1999


[40] Dinesh Varma, Sherlia Shi, Marcel Waldvogel, Dimitrios Pendarakis, ”ALMI: An Application Level Multicast Infrastructure”; Year 2000-2004

[42] Jiangchuan Liu; Rao, S.G.; Bo Li; Hui Zhang; "Opportunities and Challenges of Peer-to-Peer Internet Video Broadcast"; Proceedings of the IEEE; Volume 96, Issue 1, Jan. 2008 Page(s):11 - 24

[43] Yong Liu, Yang Guo and Chao Liang; "A survey on peer-to-peer video streaming systems"; Journal: Peer-to-Peer Networking and Applications; Springer New York; Volume 1, Number 1 / March, 2008: pp 18-28


[45] Lisa McGarthwaite; "Client-Server versus Peer-to-Peer Architecture: Comparisons for Streaming Video"; Saint Mary’s University of Minnesota


[54] Adobe Flash Media Server 3.5; URL: http://www.adobe.com/products/flashmediaserver/pdfs/fms3.5_wp_ue.pdf; Last access: 2009-07-16


[63] The Official Microsoft Silverlight site; URL http://silverlight.net/getstarted/overview.aspx; Last access: 2009-07-20


[69] FFmpeg Documentation; URL: http://ffmpeg.org/ffmpeg-doc.html; Last access: September 2009

[70] FFmpeg Server Documentation; URL: http://ffmpeg.org/ffserver-doc.html; Last access: September 2009

[71] FFmpeg Play Documentation; URL: http://ffmpeg.org/ffplay-doc.html; Last access: September 2009


[75] MYSQL; URL: http://www.mysql.com/; Last access: September 2009


[77] Sun Microsystems, Inc,"Remote Procedure Call RFC 1050"; April 1988

[78] Remote Procedure Call; URL: http://www.cs.vu.nl/~ast/books/ds1/02.ps; Last access September 2009


[82] Video Lan Media player; URL: http://www.videolan.org/; Last access: September 2009
