The Tista and Pycast systems: Streaming with open standards

Audun Vaaler
Østfold University College
School of Computer Science

Halden, May 2005

Abstract

This document describes the Tista on-demand streaming server and archiving system, and the Pycast real-time streaming encoding application, both of which have been developed at Østfold University College, Norway. It also includes a brief history of streaming, as well as an overview of significant media codecs, formats, transports and streaming protocols.

Tista is an HTTP-based server for streaming MP3 audio on demand. It differs from similar applications by supporting archiving of continuous audio streams (typically radio); by allowing extraction of clips in an easy manner (start and stop times are specified in the URI); and by scaling to a relatively large number of simultaneous listeners (700 or more, limited by hardware). Tista has been used successfully in a production environment at the Norwegian Broadcasting Corporation (NRK).

Pycast integrates various components (e.g. Lame, OggEnc, SoX and Icecast) for encoding and streaming real-time MP3 and Ogg Vorbis streams, and simplifies the administration of multi-channel, multi-bitrate and multi-format streaming systems. It lends itself well to automatic parallelisation using openMosix. Pycast has been used extensively in streaming projects at Østfold University College.

Both Tista and Pycast are implemented in the Python programming language and run on Linux servers.

Preface

About this document

This paper constitutes the author's *candidatus scientiarum* thesis, presented at Østfold University College, School of Computer Science.

Acknowledgments

I would first of all like to thank Professor Børre Ludvigsen for his guidance, support, creativity, wisdom, and never-ending supply of interesting projects, gadgets and interesting ideas.

I am also very grateful to Gunnar Misund who has provided essential feedback on this thesis, while displaying admirable patience. May your maps become infinitely detailed!

My work on Tista, Pycast and streaming in general would have been impossible without the efforts of my colleagues, both current and past. The list includes such distinguished hackers as Andreas Bergstrøm (no Linux distro is complete without the smell of garlic); Nils-Odd Solberg (democracy will never be the same again); Marte L. Horne; Håvard Rast Blok; and Thomas F. Malt (congratulations on the successful fork!). Our local administrative staff, in particular Inger-Lise B. Andersen, deserve praise for their extraordinary helpfulness, and for always keeping things running smoothly.

I would also like to thank high-flying philosopher Torkel M. Jodalen for his friendship, and his participation in innumerable expeditions both large and small. Life both analogue and digital would have been considerably less interesting without the spontaneous, creative outbursts of Halvor Kise Jr., or the experiments of Christian Raspotnig, both culinary, *ad lib*, *ad hoc* and otherwise. Siv Hilde Houmb achieved the remarkable feat of leading the way for the rest of us, in remarkably little time, while continuously smiling, running, skiing, swimming and realising that someone else had eaten her candy. Tomas Olaj, breeder of highly symmetrical dogs, should be commended for his works in the ranks of Tux, and for the century's cheesiest CS thesis.

Last, but not least, I would like to thank my family for their support, and for letting me pursue all I found interesting.

Original programming

The version of Tista presented in this paper (the source code of which is referred to in Appendix B) was implemented in its entirety by the author, with a few exceptions which are cleary marked in the source code.

Pycast has been in constant development, and includes contributions by Andreas Bergstrøm. However, the fundamental structure of the application, the original idea and the majority of the code are the work of Audun Vaaler.

Structure

This document contains six chapters. Here is an overview of their contents.

Introduction

Chapter 1 gives a brief introduction to streaming technology and terminology, and also describes Pycast's and Tista's roles in a streaming system.

The history of Internet streaming

Chapter 2 summarises the history of streaming on the Internet, beginning with Douglas Engelbart's NLS demo in 1968, and early experiments with audio conferencing on the ARPANET.

It also outlines the development of IP multicast, the MBone and CU-SeeMe, and the emerging market for IP streaming in the mid-1990s. The chapter ends with a look at streaming's recent history, and explains the roles of Tista and Pycast.

Streaming formats and protocols

Chapter 3 describes some of the most common technologies used for Internet streaming and delivery of video and audio material, and serves as an introduction to the terminology of chapters 2, 3 and 4.

It begins with a brief history of the MPEG group and the standards it has produced (primarily MPEG-1, MPEG-2, MPEG-4 and MP3). Thereafter follows a summary of standards associated with the Xiph.org Foundation, focusing on Vorbis and Ogg. The review of coding standards is concluded with a discussion of various proprietary technologies.

The second part of the chapter discusses common streaming protocols, including HTTP, RTP, RTSP, SAP and SDP, and mentions some proprietary ones.

The Pycast encoding system

Chapter 4 describes the Pycast encoding system. First, the motivation for creating the application is explained. Then the programme itself is outlined, including its purpose, overall structure and main components.

Pycast's various modules play an important role, and representative selection are presented.

The chapter concludes with a look at Pycast's suitability for automatic process migration using openMosix, as well as performance issues and potential for further development.

The Tista streaming server

Chapter 5 describes the Tista streaming server, and begins with a summary of the projects that preceded it and the experience gained.

Section 5.2 discusses the motivation for creating Tista itself, the requirements that were set, the design choices that were made, as well as details of the implementation.

The chapter ends with a look on the results achieved (including some of Tista's unique aspects), and suggestions for further development.

Conclusion

Chapter 6 concludes by suggesting directions for further development of Pycast and Tista, and also makes some observations on the state of Internet streaming and reflects on its future.

Contents

List of Figures

Chapter 1

Introduction

This chapter is a brief and relatively informal introduction to streaming technology and terminology. It also describes Pycast's and Tista's roles in a streaming system.

1.1 About streaming

Streaming is the transportation of a collection of data over a network in such a way that it can be used by the receiver (or receivers) before the entire transfer has finished.

For instance, with streaming video it is possible to watch an event in real-time (as it happens), to start watching a recording (a video file) before all of it has finished downloading, or to talk face-to-face with people all over the world.

1.2 Acquiring data

Any streaming system needs one or more data sources from which to obtain data. A real-time streaming system, such as the combination of Pycast and Icecast [1], will use a live feed, for instance a radio receiver. (A purely Internet-based stream would be directly connected to some kind of production system.)

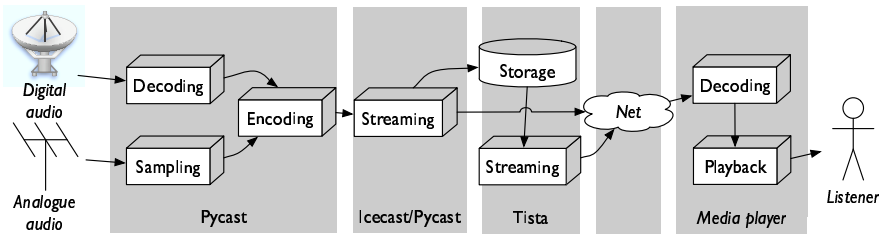

Figure 1.1: Components of a Pycast/Icecast/Tista streaming system

If a data source is analogue (for instance an FM radio) it must first be *sampled* to convert it to digital form. Audio from digital sources has been subjected to sampling already, but often need to be *transcoded* from one digital format to another.

1.3 Encoding and transcoding

Before being distributed onto the network a stream must be *encoded* into a specific format, so that it is easily understandable for media players[1]. Encoding usually also involves *compression* of the data to conserve bandwidth.

The most common audio compression algorithms (such as MPEG-1 Layer 3 [2]) are *lossy*, and achieve high compression ratios by discarding nuances that are not easily perceptible to human brains, ears and eyes. Algorithms that preserve all of the information inherent in the data (for instance FLAC[3]) are *non-lossy*.

Transcoding is the conversion of data between different formats and/or quality levels. For instance, Pycast is often used to transcode high-bitrate MPEG-1 Layer 2 (MP2) streams to lower-bitrate MP3 streams. The number of transcoding steps between lossy formats should generally be kept low, to avoid unnecessary loss of quality. Transcoding is usually achieved by first decoding the source data to an uncompressed intermediary format, which is then re-encoded into the target format.

1.4 Streaming

After encoding, the data is ready for distribution to listeners or viewers, possibly after being stored a while for on-demand playback. *Streaming* is often performed by a dedicated streaming server (e.g. Icecast), but is sometimes also done by the encoding system itself. (For instance, Pycast can transmit streams via IP multicast (see section 2.3 without the aid of a separate streaming server.)

1.5 Reception and playback

On a user's computer, streams are usually played back with a stand-alone media player (such as VLC [2], QuickTime Player [3] or Windows Media Player [4]), or embedded inside a Web page in a browser.

The term *streaming* implies that a stream is played back while it is being received. It is also common to download a stream in its entirety before playing it.

[1] Please note that the term *media player* is used in the widest sense to refer to any application capable of playing and presenting video and audio, and not a specific product, such as Microsoft's Windows Media Player.

[2] See section 3.1.2.

[3] See section 3.2.1.

1.6 Pycast and Tista

Pycast and Tista can be combined with Icecast to make a complete real-time and on-demand radio streaming system (figure 1.1). In that case Pycast will be used to acquire (from digital or analogue sources) and encode/transcode audio streams. It is also capable of transmitting streams via IP multicast.

The version of Tista described in this paper is not capable of streaming MP3 data in real-time. It is therefore necessary to enlist the help of an Icecast server to distribute the streams to listeners.

Tista also uses the Icecast server as a source for continuously downloading and storing streams, which are instantly made available for on-demand listening or downloading.

Chapter 2

The history of Internet streaming

This chapter gives an overview of the history of streaming (and other forms of media delivery) on the Internet (figure 2.1), and relates this to the development of Tista and Pycast.

2.1 NLS

It is difficult to summarise any area of personal computing without mentioning the work of Douglas Engelbart; this applies to streaming as well.

Engelbart's demonstration of the *online system* (NLS) [5, 6] at the 1968 Fall Joint Computer Conference in San Francisco introduced such concepts as the mouse, graphical user interfaces (GUIs), hypertext, and the idea of PCs itself.

The demonstration also used live video feeds communication with remote users[1]. While this can not be considered streaming (he feeds were carried with analogue signals over a circuit-switched network), it foreshadowed what was to come.

2.2 Early ARPANET experiments: NVP, NVP-II and PVP

The first experiments with streaming audio in packet-switched networks took place on the ARPANET (the precursor of today's Internet) in the early 1970s.

One important result was the Network Voice Protocol (NVP) [7], which was first implemented in December 1973. NVP was part of ARPA's Network Secure Communications (NSC) project, which sought to "develop and demonstrate the feasibility of secure, high-quality, low-bandwidth, real-time, full-duplex (two-way) digital voice communications over packet-switched computer communications networks".

[1] Analogue video feeds also delivered the GUI, since the NLS computer itself was situated at the Stanford Research Institute, a distance away.

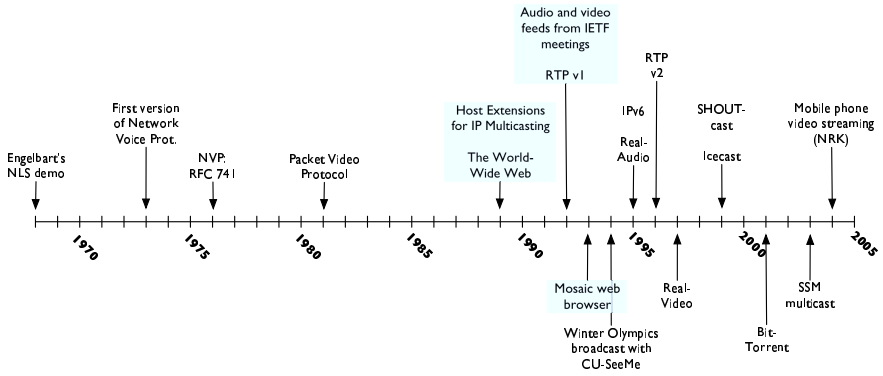

Figure 2.1: A streaming timeline

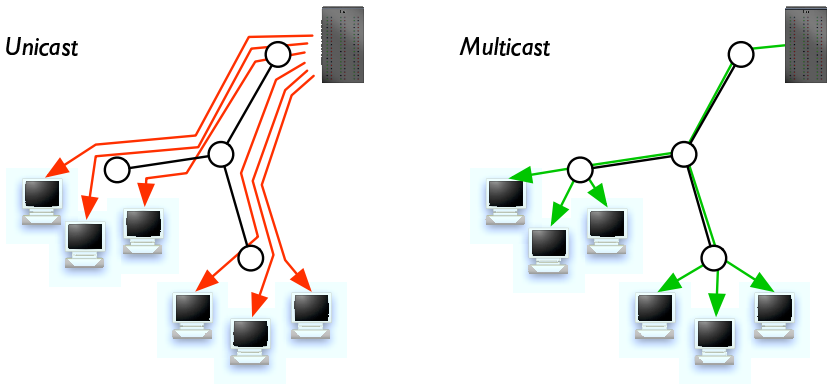

Figure 2.2: Unicast (left) vs. multicast (right) delivery. By intelligently distributing one-to-many or many-to-many data in the network, multicast can significantly reduce load.

Digital, packet-switched audio communication was (quite rightfully) assumed to be potentially very useful for the military: Such a system could be deployed world-wide, be more robust than circuit-switched telephones, and audio streams could easily be secured using encryption.

NVP was formally defined in RFC 741 [7] in 1976, and was followed by NVP-II [8] and the Packet Video Protocol [9] in 1981.

In the early 1990s NVP was adapted for use in multicast experiments [10], and strongly influenced the design of RTP (see section 3.4.3).

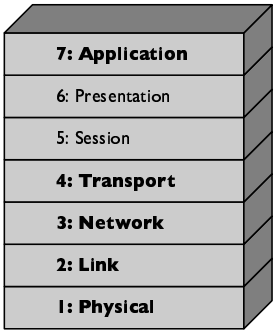

Figure 2.3: The Open Systems Interconnection Reference Model (OSI). Layers in boldface map directly to layers in the TCP/IP stack.

2.3 IP multicast and the MBone

2.3.1 The birth of IP multicast

Interest in streaming and other media delivery over the Internet seems to have been quite low during the 1980s, probably due to limited bandwidth and lack of multimedia support in hardware and software.

This started to change around 1990, initially fueled by the advent of IP multicasting [11], pioneered by Stephen Deering. Data which need to be transmitted to multiple receivers (by one or more senders) can be more efficiently transported if it never travels along a specific network link more than once (figure 2.2). Schemes that accomplish this are generally referred to as *multicast* protocols.

IP multicast works on layer three of the OSI model (figure 2.3); any multicast-enabled router will therefore be able to route IP multicast traffic. Users join a multicast *group* by listening to a specific multicast IP address.

In the late 1980s and early 1990s, when bandwidths were far lower than today, IP multicast made it possible to conduct audio and video conferences and broadcasts that scaled beyond just a few participants. This led to an exploding interest in multimedia communication on the Internet.

2.3.2 The MBone

Since few routers at the time supported multicast, a virtual network based on tunnels and multicast-native routers was established. The network was later named the *MBone* (multicast backbone) [12].

In March 1992 the MBone was used to distribute real-time audio from the 23rd Internet Engineering Task Force (IETF) meeting in San Diego, USA to 20 sites world-wide. This was the first large-scale audio multicast over a packet-switched network.

As the number of MBone users rapidly grew, the amount of material available there increased as well. Streams included meetings and conferences (naturally), TV channels (e.g. NASA TV [13]), radio stations, lectures, and (of course)

the obligatory webcams overlooking a parking lot[2].

While interest relative to other technologies has waned somewhat, the MBone is still an interesting place, with lots of high-quality feeds. A visit is recommended to anyone with an Internet connection sophisticated enough to support it.

2.3.3 Problems

For almost half a decade multicast streaming remained a popular and promising area, and was assumed to soon move into mainstream use, together with other Internet technologies such as the World-Wide Web. Unfortunately, unforeseen difficulties have slowed down the proliferation of multicast significantly. [14] mentions several reasons, including these:

- Lack of protection against attacks by unauthorised senders.
- High implementation complexity.
- Problematic address allocation, including relatively high risks of address collisions.
- Poor scalability when routing multicast data between organisations.

2.3.4 Source-specific multicast

An encouraging and relatively recent development is source-specific multicast (SSM) [15], which in contrast to the original multicast approach (referred to as any-source multicast, or ASM) only allows only one sender in a multicast group: Instead of being defined only by its IP address, an SSM group is defined by the combination of its IP address and the IP address of the sender.

SSM is therefore ideal for one-to-many streaming (e.g. TV or radio), and many of the problems associated with ASM no longer apply:

- Since groups have only one sender, there can longer be such a thing as unauthorised transmitters. (Group members explicitly limit which sender they will receive data from.)
- Address allocation becomes trivial, since different senders may use the same target IP addresses without interfering with each other.
- According to [14] complexity and routing difficulties are significantly reduced.

IP multicast, especially SSM, is probably due for a comeback in the near future, since it is potentially very valuable for truly large-scale distribution of real-time data. (Ideally, the Internet should be able to e.g. handle real-time streaming of high-quality, real-time video to millions of viewers.)

[2]It seems that one of the first applications of any new video streaming technology is to use it to broadcast the view of a parking lot, or any other view available in the nearest window

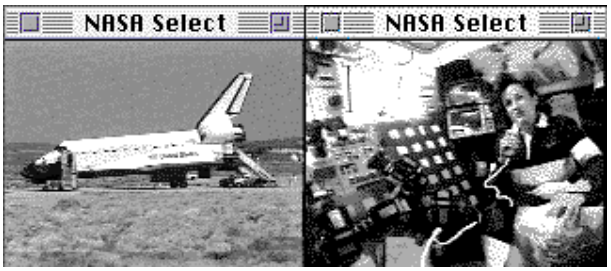

Figure 2.4: CU-SeeMe feeds from the Space Shuttle. (Screenshots by Halvor Kise Jr.)

2.4 CU-SeeMe

2.4.1 History

In the beginning the MBone tools were only available for UNIX platforms, making it impossible for users of Macs and PCs (which at the time had relatively unsophisticated networking support) to participate. Timothy Dorcey and Richard Cogger of Cornell University, consequently set forth to program an inexpensive videoconferencing tool that should be usable on computers that were then widely available.

Their theory, which turned out to be correct beyond their expectations, was that an accessible, low-cost solution would be widely and quickly adopted.

The result, CU-SeeMe [16], first appeared in the autumn of 1992, initially only for the Macintosh [17, chapter 8] and without audio. Video was monochromatic (16 shades of grey, 160x120 pixels) and compressed using a custom algorithm optimised for use on Motorola 68020 and 68030 processors. More advanced versions, complete with audio support, followed, including ports to Microsoft Windows.

2.4.2 Reflectors

Since CU-SeeMe could not rely on IP multicast for efficient stream delivery, its authors decided to ensure scalability by distributing data via *reflectors*, servers that could work as distribution hubs for multi-party conferences or broadcasts.

CU-SeeMe's reflector system may in other words be seen as an *application-level* multicasting scheme (OSI layer 7).

2.4.3 The CU-SeeMe community

CU-SeeMe very soon gained a large following of enthusiastic users, inspired by the software's ease of use, and the experience of communicating face-to-face with people all over the world. In addition to ad-hoc chats and conferences with friends, colleagues or even complete strangers, CU-SeeMe was also used to broadcast events such as space shuttle flights (figure 2.4), conferences, Nobel Prize ceremonies [17, page 23], and the 1994 Olympic Winter Games [18].

Børre Ludvigsen and Halvor Kise Jr. at Østfold University College played active roles, and were responsible for operating three high-traffic reflectors,

performing the Nobel and Winter Games broadcasts, and broadcasting *Radionettet*, the world's first regular radio programme available online, in cooperation with the Norwegian Broadcasting Corporation (NRK). Mr. Ludvigsen even took his entire home online [19].

2.4.4 CU-SeeMe today

CU-SeeMe use has been declining for the better part of a decade, and the system is currently mostly of historical interest. Its spirit flourishes, though, in the form of media streaming, instant messaging (IM) clients with video conferencing support, as well as a plethora of webcams keeping eyes of everything from parking lots to goldfish.

Timothy Dorcey continues to work with video conferencing technology as part of iVisit [20].

2.5 RealAudio

Internet streaming took its first real steps into the mainstream in the mid-1990s, when there was a surge of interest in streaming solutions that could be easily integrated with the Web. As always in an emergent market several contenders appeared, including Progressive Networks, VDOnet (makers of VDOLive), Xing (makers of StreamWorks), Liquid Audio, and AudioActive [21]. Few of the companies have survived; the rest have either been merged into other companies, changed their business focus, or gone bankrupt. Progressive Networks did better.

In April 1995 [22] Progressive launched RealAudio, a codec and streaming server solution for audio that became very popular very quickly. The reasons for RealAudio's success are not immediately clear a decade later; two likely explanations are that it was among the first to offer actual streaming (playback during download), and that it had advanced features such as user-friendly seeking. (Competing products seem to have had surprisingly Spartan user interfaces.)

In 1997 Progressive was renamed RealNetworks, and introduced the RealVideo codec as part of RealPlayer 4.0. Real today remains one of the three major commercial streaming suites, together with Microsoft's Windows Media and Apple's QuickTime.

Windows Media and QuickTime did not support streaming until the spring of 1999, when version 4 of both products was released [23, 24].

2.6 Shoutcast and Icecast

2.6.1 The MP3 phenomenon

Towards the end of the 1990s MPEG Layer 3 audio (MP3, see section 3.1.2) began to emerge as a grass-roots alternative for storing and sharing music. First of all it was capable of efficiently compressing audio files with only a minimal loss of quality. Secondly, the number of computers capable of MP3

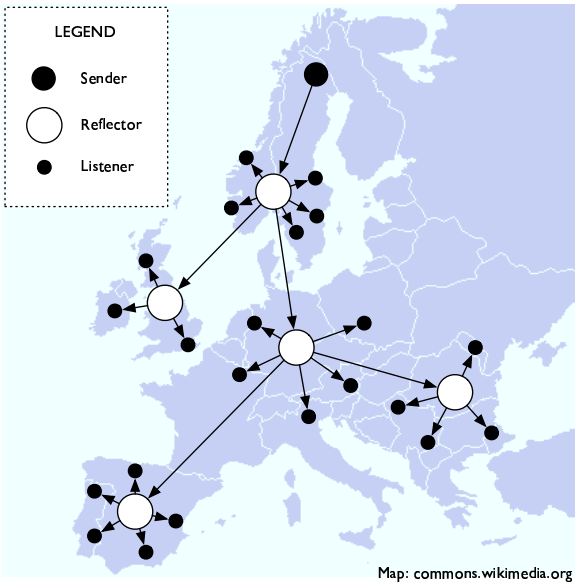

Figure 2.5: SHOUTcast/Icecast reflectors. The large black dot is a sender; the smaller black dots are listeners; the large white circles are reflectors. (This is an illustration only, and does not show an actual reflector constellation.)

playback had reached a critical threshold. Also, the format was open, with a wide selection of players, encoders and other tools that supported it.

The MPEG standards had been designed with streaming in mind, and MP3 consequently lends itself well to being streamed; indeed creating a stream was as simple as placing an MP3 file on a web server and listening to it with a player that supported progressive downloading[3] of media (for instance WinAmp [25]).

2.6.2 MP3 streaming

Two groups took advantage of this: Nullsoft (the creators of WinAmp) launched *SHOUTcast* [26] in December 1998 [27]; Icecast [1], a free software project, appeared shortly thereafter.

The two applications were quite similar: Both streamed real-time MP3 audio over HTTP, and could be used with most MP3-compatible media players.

Also, both supported the concept of reflectors (yet another instance of application-level multicast) for scalability: A sender would typically send his or her stream to a remote SHOUTcast or Icecast server, which distributed it either directly to listeners, or via one or more other servers acting as a reflectors (figure 2.5). This freed would-be broadcasters from having to run their own streaming servers, and allowed even home users with very limited upstream bandwidth to share their own radio programmes with the world.

[3] Progressive download occurs when a media player is able to start playback while a file is being downloaded.

Another important advantage was that MP3 and HTTP offered better audio quality and supposedly fewer connection problems [28] than the competition, which at the time mostly consisted of RealAudio. (To be fair, MP3 was probably streamed at higher bitrates, and the TCP-based HTTP brings with it higher traffic overheads than e.g. UDP-based streaming.)

2.6.3 Webcasting today

HTTP-based web radio (usually operated with SHOUTcast or Icecast servers) has remained very popular. *Webcasters* include large radio stations like Virgin Radio in the United Kingdom [29], and the Norwegian Broadcasting Corporation, in co-operation with Østfold University College [30].

2.7 IPv6

The Internet Protocol, version 6 (IPv6) [31] was first published in 1995 and improves the currently widespread IPv4 in many ways that are useful for streaming and media delivery.

Most importantly, it vastly expands the IP address space (by a factor of 2^{96}), making it possible to connect a virtually limitless number of devices of all kinds to the Internet. This will hopefully reduce the temptation of using network address translation [32], strengthen end-to-end connectivity, and make peer-to-peer distribution of data more practical and efficient.

Secondly, IPv6 includes native support for multicasting, including SSM. IPv6-enabled network should therefore be able to transport realt-time data much more efficiently than today.

Finally, IPv6 also supports prioritisation of packets, making timely delivery of time-critical data easier.

2.8 Recent history

Streaming on the Internet can trace is history back more than 30 years, but has not been part of the computing mainstream for more than five to ten. Between 1995 and 2000 it bore the hallmarks of an emerging industry, with a large number of start-up companies competing for domination.

During the last five years it has been dominated by consolidation (devouring most of the early start-ups) and maturation. The dominant streaming platform today is Microsoft's Windows Media, in line with developments elsewhere in the IT industry. More people create, listen to and watch streams than ever, but the pace of innovation has inevitably slowed down.

On the other hand many interesting things have happened, especially on the grass-root level. For instance, the BitTorrent peer-to-peer (P2P) system [33, 34] has made it possible for persons and small companies to distribute large files (of megabyte or gigabyte size) to hundreds or thousands of users. (Data transfer at such scales have traditionally required more bandwidth that individual users could not afford.)

Interesting tendencies to watch in the future include P2P publishing systems; delivery of media to mobile computers (including cell phones [4] and MP3 players); open standards versus increasing demands for control (including digital rights management); and the possible revenge of IP multicasting.

2.9 Other technologies

Several important technologies (e.g. MPEG) have received little or no attention in this chapter, but will be covered in more detail in chapter 3.

2.10 Where Tista and Pycast fit in

Both Tista and Pycast were created to solve problems for which suitable tools did not exist.

An archiving and on-demand streaming system like Tista could for instance have been implemented by combining a scheduled recording programme with a regular web server. However, such a solution brings with it flexibility and scalability problems (see section 5.1.3.

Tista does nothing new in the areas of streaming protocols or formats, but has innovative features for continuous stream archiving and user-friendly and very flexible extraction of clips. It also scales to a larger number of concurrent users than most web servers do (see section 5.2).

The *de facto* standard for encoding Icecast streams is Ices [36]. Pycast, however, offers a few potentially very useful extra features: Most importantly, it can use DVB streams for audio input (reducing the need for radio tuners), and can easily handle a large number of streams (encoders run as separate processes). See section 4.2 for more information.

[4] NRK started streaming TV to mobile phones in 2004[35]

Chapter 3

Streaming formats and protocols

Media coding standards and transport protocols are essential for streaming. This chapter describes some of the most common technologies, including MPEG, Ogg, HTTP, RTP and RTSP.

3.1 MPEG standards[1]

All of our streaming activities involve MPEG standards, either at the source (e.g. when transcoding from a digital video broadcasting (DVB) or digital audio broadcasting (DAB) stream), when streaming to clients (e.g. MPEG-1 Audio Layer 3 (MP3) radio streams) or both. This section outlines the history of the MPEG standards with an emphasis on the specific technologies we use.

3.1.1 Background

The Motion Pictures Expert Group (MPEG) was formed in 1988 to standardise rapidly evolving video encoding technology. Video naturally played an important role in the emerging multimedia industry. Computer companies as well as the home electronics and entertainment industries mainly concentrated on off-line digital storage based on CD-ROM; telecom and broadcast companies needed video coding solutions for teleconferencing and digital TV distribution.

The MPEG group has since released three comprehensive standards for audio/video coding and encapsulation: MPEG-1 (1992), MPEG-2 (1994) and MPEG-4 (1998).

3.1.2 MPEG-1

MPEG's initial focus was CD-ROM-based applications, with a target bitrate of 1.5 Mbits/s (slightly above the 150 kbyte/s transfer rate of a 1x CD drive[2]).

[1] This section is based on [37, 38, 39].

[2] The speed of a CD drive is usually indicated using multiples of the 150 kbyte/s base rate, e.g. 2x (300 kbyte/s) or 10x (1500 kbyte/s).

Since there was no need to encode video in real-time, technology designers could prioritise quality over encoding speed.

The MPEG-1 standard defines three audio codecs of increasing complexity, namely MPEG-1 Audio Layer 1, 2 and 3 (usually referred to as MP1, MP2 and MP3).

Since MPEG-1 was designed to be used in settings with low error rates, MPEG-1 streams are not very resilient to data loss. Audio and video tracks (in the form of packetised elementary streams (PES)) are multiplexed into programme streams (PS) for storage. The standard also does not support interlaced coding of video.

3.1.3 MPEG-2

The MPEG-2 standardisation process began in 1990 with the goal of creating a generic video compression system suitable transmissions such as TV broadcasting (satellite, terrestrial and HDTV), radio broadcasting and videoconferencing. MPEG-2 is based on MPEG-1, and MPEG-2 decoders are required to support playback of MPEG-1 streams with MP1, MP2 or MP3 audio.

In addition to extending MP3 with three new sampling rates, MPEG-2 also introduced the Advanced Audio Coding (AAC) codec, which significantly more efficient codec at the cost of backwards compatibility. (According to [38] AAC yields the same quality as MP3 at around 70% of the bitrate.)

Since MPEG-2 was to be used in a much wider range of settings: In addition to supporting reliable media like CD-ROM and DVD discs, MPEG-2 streams were required to work well when streamed across asynchronous transfer mode (ATM) networks and to support multiplexing of streams with different time bases[3].

MPEG-2 therefore defines two different multiplexing schemes: Programme streams (PS) provide backwards compatibility with MPEG-1 PS streams. Transport streams (TS) make it possible to multiplex PES streams with different time bases. Also, the small TS packet size[4] of 188 bytes reduces the impact of packet loss, and facilitates use of forward error correction. The TS format is therefore well suited for use in noisy or lossy contexts (e.g. satelitte broadcasting).

It should be noted that in some cases (e.g. for RTP streaming, see section 3.4.3) individual PES streams are transmitted using separate channels provided by the underlying transport protocol (e.g. IP) instead of using PS or TS multiplexing.

MPEG-2 also added support for encoding of interlaced video (for TV and traditional video compatibility), as well as higher bitrates (beyond 10 Mbits/s) and larger picture sizes.

Some current mainstream uses of MPEG-2 include digital TV broadcasting (e.g. DVB and HDTV) and video storage on DVDs.

[3] Related media streams (e.g. the audio and video tracks of a TV channel) use the same time base so that they can be properly synchronised during playback. Often it is also desirable to multiplex unrelated streams (e.g. when transmitting several different TV channels via one physical satellite transponder), in which case it is impractical to use one single time base.

[4] In this context a packet is a portion of MPEG data, and should not be confused with e.g. an IP packet. It is, however, likely that an MPEG packet would be mapped onto a corresponding packet in the underlying transport layer.

3.1.4 MPEG-4

Development of what was to become MPEG-4 started in 1992 when an ad-hoc group was formed to investigate the feasability of a 10 kbit/s audio/video codec. Formal work started began in 1993.

The initial goal was to identify far-term applications and requirements for very low-bitrate video coding. It soon became clear that the similar H.263 standard being developed by the International Telecommunication Union (ITU) would provide performance close to the technological limits of the day. The MPEG-4 group assumed that their work on another codec would do little to improve the state of the art within a self-imposed five-year limit.

Instead of dissolving the group, focus was moved to an analysis of trends in the audio/video (AV) world, based on the convergence of TV, movies, entertainment, computing and telecommunications. Important goals became support for new kinds of AV communication, access and manipulation; a more advanced bitstream (multiplexing) format; improved efficiency (for low-bitrate communication or low-capacity storage); and video with better subjective quality than H.263.

In addition to providing AV codecs and an advanced multiplexing system MPEG-4, also supports integration of other data types like text, graphic overlays, synthetic music, spatialised audio, irregularily shaped video frames, 3D virtual worlds and hyperlinks. Because of the standard's size, creators of MPEG-4 software are not expected to implement the entire specification, but to choose useful components as needed.

MPEG-4 provides two file formats: XMT (extensible MPEG-4 textual format) stores data using relatively abstract, high-level, XML-based structures, which makes XMT files suitable for further processing in e.g. a video editing programme. In contrast, the MP4 format is more rigid, and leaves less control to users (since the streams are not supposed to be edited or changed). MP4 files are used for distribution of finished MPEG-4 content.

Additionally, MPEG-4 specifies two standards for streaming, one based on RTP for use in IP networks, and one for carrying MPEG-4 data using MPEG-2 transport streams.

Using a selection of video codecs MPEG-4 supports video bitrates from 5 kbit/s to 1 Gbit/s. MPEG-4's speech coding supports rates from 2 kbit/s to 24 kbit/s (or a lower limit of 1.2 kbit/s using variable bitrate coding). Its general audio codec (mostly based on MPEG-2 AAC) supports bitrates from 4 kbit/s and up.

3.1.5 MPEG-3, MPEG-7 and MPEG-21

Three other MPEG standards exist. Since none of them specifies video or audio compression standards, I will describe them only very briefly here:

- The purpose of MPEG-3 was to develop compression standards for high-definition TV (HDTV). When it became clear that MPEG-2 codecs were sufficient, work on MPEG-3 was stopped and MPEG-2 adopted for HDTV applications.
- MPEG-7 is a standard for describing multimedia content (including pictures, video, speech, real and synthetic audio as well as 3D models), in a

way that is independent of formats and storage media (including paper, film and tape).

- The goal MPEG-21 is to investigate the different parts of the multimedia production and distribution chain, and to define appropriate standards for the various processes involved.

MPEG-7 and MPEG-21 are not yet widely used.

3.2 Xiph.org standards

3.2.1 Background[5]

In 1998 the Fraunhofer Institute announced that it would begin to charge fees from users of the MP3 codec (including developers of alternative implementations). Alarmed by this development, free software programmer Christopher Montgomery began work on an alternative, patent-free codec. He was joined by other programmers, and version 1.0 of the codec, named Vorbis, was released in 2002. (Several open-source and independent MP3 encoder implementations later fell victim to the stricter enforcements of Fraunhofer's intellectual property rights.)

Development is now coordinated by the Xiph.org Foundation (formerly known as Xiphophorus), along with the related (at least in spirit) Theora, Speex, FLAC, Tarkin and Writ standards.

Vorbis is formally known as Ogg Vorbis[6]; Ogg is an encapsulation standard used by many of the formats mentioned above.

3.2.2 Vorbis

As already mentioned Vorbis was developed to fill the same roles as MP3, and shares many of the same characteristics. Apart from Vorbis's slightly better compression rates and some novel technological features (like bitrate peeling[7]) the main differences are related to patents and licensing.

The Vorbis standard itself has been placed in the public domain, and is unencumbered by patents or licensing issues[8]. The Xiph.org foundation has, however, reserved the rights to set further Vorbis specifications and to certify standards compliance.

Libraries developed as part of the project are covered by a BSD-style license; tools are covered by the GNU General Public License.

Ogg Vorbis is widely supported by many media players, including hardware-based and portable ones, but still lags behind MP3.

[5] This section is based on [40, 41].

[6] The name Ogg is supposedly taken from the Netrek computer game[42]. Vorbis is named after a character in the novel *Small Gods* [43], part of Terry Pratchett's *Discworld* series [41].

[7] Ogg Vorbis supports the rather interesting concept of bitrate peeling: An encoded stream may consist of layers of data at different levels of detail. Instead of encoding several streams at different bitrates (e.g. to cater for different types of network connections), one stream can be encoded instead and layers later peeled away until the resulting stream has a low enough bitrate. This feature has not been implemented by any encoding tools yet.

[8] Some people external to the project claim that Vorbis is not entirely patent-free.

3.2.3 Ogg

The Ogg format is used for encapsulation and transport of Vorbis streams and many of the other formats developed under the Xiph.org umbrella. Version 0 of the standard is documented in RFC 3533 [44], which also specifies an RTP transport scheme.

An encapsulated stream (a *physical bitstream* in Ogg terminology) contains one or more *logical bitstreams* produced by one or more encoders. Logical bitstreams are delivered to the Ogg multiplexer in *packets* (not to be confused with network packets) and placed in *pages* equipped with headers. The multiplexer pays no attention to the actual contents of logical streams.

All logical streams start with a beginning of stream (BOS) page that identifies the codec in use and usually contains information about the encoded stream (e.g. sample rate and number of audio channels).

3.3 Other standards

Coding standards are legion; I will mention a few more:

- Windows Media [4] is a multimedia framework developed and used by Microsoft, and is included in most modern versions of Windows. Due to the current software market situation it can be found on most PCs. While it supports some open standards like MP3 and MPEG-1, its more advanced codecs, as well as the encapsulation format, are proprietary and closed.
- QuickTime [3] is a multimedia framework developed and used by Apple, and is included in all current versions of Mac OS. It supports several open standards (e.g. MPEG-1, -2 and -4, as well as MP3 and AAC), but also includes many proprietary ones. MPEG-4's MP4 format is based on the QuickTime encapsulation system.
- RealNetworks [45] played an important part in developing Internet streaming software in the early 1990s, and now have a large numbers of users of its RealMedia, RealAudio and RealVideo products. Several of their codecs and other technology have been open sourced and are currently developed as part of the Helix project [46].

A very interesting recent development is BBC's Dirac [47] codec, which aims at general high-quality, high-efficiency coding of video at resolutions from 180x144 to 1920x1080 pixels, with a minimum of patent liabilities. Dirac is a part of the BBC's effort in investigating and developing technologies for large-scale distribution of media via the Internet, and is developed using an open source model.

3.4 Streaming protocols

3.4.1 A note on multicast

The term multicast in its most general sense describes any protocol for efficient delivery of data to multiple network destinations, ideally in such a way that

```
HTTP/1.1 200 OK Date: Fri, 17 Sep 2004 22:48:22 GMT
Server: Apache/1.3.33 (Darwin)
Last-Modified: Fri, 17 Sep 2004 22:46:25 GMT
ETag: "ecada-70434-42151ec1"
Accept-Ranges: bytes Content-Length: 459828
Content-Type: application/ogg
```

Figure 3.1: Typical header returned by an Apache web server when a file has been requested. Note that both modification date (`Date`) and file size (`Content-Length`) are included.

```
HTTP/1.0 200 OK
Content-Type: application/ogg
ice-audio-info: samplerate=48000
ice-name: NRK Alltid Klassisk
ice-public: 1
Server: Icecast 2.0.0
```

Figure 3.2: Typical header returned by an Icecast HTTP streaming server when requesting a stream. (Some of the header lines represent Icecast extensions.)

each copy of a given piece of data travels across a specific link only once.

Please note that the term is used more specifically in this document to signify IP multicast (either any-source or source-specific).

3.4.2 HTTP

The Hypertext Transfer Protocol (HTTP) [48] is not specialised for streaming, but still deserves to be counted as one of the most important streaming protocols, mostly because of its ubiquity, familiarity and simplicity.

Since HTTP is quite well-known, I will only cover it quite briefly.

Streaming with HTTP

HTTP was, as its name suggests, designed for transferring hypertext files over the Internet. When used for streaming it facilitates the transportation either of files in such a way that they can be played back while downloaded, or of continuous streams with no definite length (e.g. a radio broadcast). Figures 3.1 and 3.2 show the headers returned by web servers in two typical cases.

Most media players[9] support such progressive playback of streams. However, some media files are organised in ways that make them unsuitable for HTTP streaming. For instance many MPEG-4 need to be fully downloaded before playback can start, since essential information is found at the end of the file.

[9] Players on current mobile phones seem to represent a notable exception.

Advantages

HTTP as a streaming protocol has several important advantages, mostly due to its familiarity and ubiquity.

- Streams can be treated just like files: Since HTTP is a general data transfer protocol any application or tool that can download a file can also download streams. This means, among other things, that users and programmers can reuse existing software and knowledge, thus lowering the threshold for further innovation.
- Increased transparency: Because HTTP is widespread, well-known and technically quite simple, using it makes the workings of streaming software easier to understand, learn and experiment with.
- Increased predictability: HTTP is one of very few protocols that can traverse the obstacles of today's Internet with relative ease. Many other protocols (and sometimes even HTTP) encounter problems for instance due to firewalls and network address translation (NAT) routers. Also, because of its essential nature, most networking equipment and software are specifically designed to work well with HTTP.
- Improved interoperability, due to the wide variety of programmes that support HTTP.

Disadvantages

While HTTP often works well, it is not perfect for streaming. Designed for transferring files, not streams, it is both less efficient and has fewer specialised features than many dedicated streaming protocols.

- HTTP is less efficient than most other streaming protocols, since it is connection-oriented and based on TCP. With a UDP-based stream lost packets can be skipped (often with minimal consequences for listeners or viewers); using a TCP-based stream implies the added overhead of keeping track of and resending lost data.
- Lack of streaming-specific functionality: HTTP is not designed for media streaming and does not support useful functions such as random-access seeking[10].
- Poor scalability: HTTP is inherently unicast-only, and scaling to a large number of simultaneous listeners (especially at high bitrates) is not trivial. One possible workaround is to use a network of relay servers.
- Streams can be treated just like files. This means that it is significantly easier to obtain a perfect copy of an HTTP stream than e.g. one streamed

[10]HTTP does, however, support resuming of downloads: Typically, a client that has been interrupted while downloading a file can request it again but indicate that it is interested in retrieving only a part of the file (given as a range of bytes). This mechanism is used by some media players (e.g. Winamp [25]) to implement basic stream seeking capabilities, by approximating byte offsets based on the average bitrate of a stream.

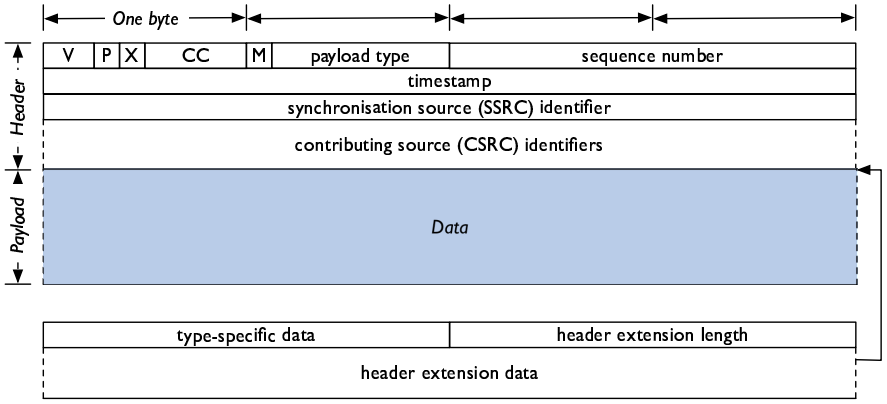

Figure 3.3: RTP packet structure. The extension header is optional.

using UDP. Tools for capturing HTTP streams are widely available (indeed, any web browser will suffice), and the use of TCP guarantees a perfect copy without missing data.

(On the other hand, any stream published or broadcast in any medium or using any protocol can be recorded. HTTP just makes the process a bit easier.)

3.4.3 RTP

The Real-time Transport Protocol (RTP) [49] specifies a standard packet format for transporting media streams on packet-switched networks. It is commonly used for distributing video and audio streams on IP networks (in real time or on demand) and for conferencing. RTP is suitable for both unicast and multicast distribution, and can run on top of both UDP and TCP. It may be used with a wide variety of codecs; specifications exist for carriage of both MPEG-1, MPEG-2, MPEG-4, Ogg and several other streaming formats.

In contrast to HTTP, RTP is only a transport protocol, and has no mechanisms for e.g. request handling or negotation. Such tasks are usually handled by offering session description files via HTTP, RTSP or SAP [50].

Since RTP is less well known than HTTP I will outline it in slightly more detail below.

The anatomy of RTP packets

The RTP protocol specifies the format of a header that is prepended to all of the packets in a stream. The actual data in a packet is referred to as the *payload*, and usually consists of an integral number of logical units (e.g. MPEG frames). Units may, however, be split between packets.

RTP headers contain the following parameters (see figure 3.3). Numeric values are big-endian and unsigned.

Version (V) A two-bit number that identifies the RTP version (a value of two indicates an RFC 3550 packet).

Padding (P) A one-bit flag that, if set, specifies that padding bytes follow the payload. If so the last padding byte specifies the number of bytes to be ignored.

Extension (X) A one-bit flag that, if set, indicates that the main header is followed by an experimental header extension.

CSRC count (CC) A four-bit number that specifies the number of contributing source (CSRC) identifiers at the end of the header.

Marker (M) A one-bit flag that, if set, marks the packet as exceptional; the exact meaning depends on the type of data being streamed. For instance, when streaming MPEG video a set flag indicates that a packet contains the rest of a frame that had to be split between several packets.

Payload type A seven-bit number that specifies the payload type. For instance, RFC 1890 [51] defines payload type numbers for several codecs and data types, including MPEG audio (14) and MPEG video (32).

Sequence number A 16-bit number that is incremented by one for each packet sent, enabling receivers to easily sort data in the right order and detect packet loss. (After it reaches it maximum value (65535) it wraps around to zero.)

Timestamp A 32-bit number that represents the time when recording of the payload started, making it possible for receivers to synchronise multiple streams and to detect network jitter (the variance in packet transit times). The timestamp's resolution depends on the type of data being streamed. Since different data types (e.g. MPEG video and MPEG audio) often use timestamps with different resolutions, related streams are synchronised using out-of-band RTCP packets (see 3.4.3).

Synchronisation source (SSRC) identifier A 32-bit, randomly chosen number that uniquely identifies a stream source. (The source address of a packet is not necessarily a unique ID. For instance, sources located behind NAT gateways may happen to share the same IP private IP address[11].)

Contributing source (CSRC) identifiers A row of 0 to 15 32-bit numbers that indicate the SSRC IDs of the sources that contributed to a packet. (RTP supports entities called *mixers*, for instance placed at firewalls between networks, that combine and forward packets. Packet payloads are sometimes also transcoded to other formats. In a conference setting it is often desirable to identify speakers and other contributors; CSRCs make this possible. If the number of contributors exceeds 15 some will not be credited.)

If the extension flag is set the main header is followed by an extension header containing the following fields:

[11] See [32, 52] for more information.

Type-specific data 16 bits whose meaning is dependent on the type of data being transported.

Header extension length A 16-bit number between 0 and 65535 that specifies the number of 32-bit extension data words that follow.

Header extension data Optional data associated with the extension header.

Extension headers are used for protocol experiments where experimental data need to be placed in the header; it is otherwise recommended to use the payload itself for such purposes.

RTCP

RTP receivers are expected to send reception feedback to all other participants in a session, using the RTP Control Protocol (RTCP), to aid flow and congestion control, diagnose distribution problems and for adjusting adaptive codecs. RTCP is also used for management tasks such as keeping track of participant IDs in case of conflicting SSRCs, and for providing the necessary data to synchronise related streams (e.g. video and audio tracks).

To make scalable distribution possible the amount of RTCP traffic is limited by making participants adjust their RTCP transmission rates such that the total rate stays below a certain threshold (usually 5% of the streams themselves).

RTCP can be used for transporting simple session control information, such as participant names.

Source-specific multicast

Source-specific multicast (SSM) [15] is an IP multicast scheme that differs from traditional any-source multicast (ASM) in that a multicast group is defined by the source IP address as well as the target multicast group; in other words a group can contain only one source. SSM is therefore well-suited for one-to-many applications such as video and audio streaming.

No standards currently define the RTCP communication from receivers to senders in SSM settings; receivers should therefore not send any RTCP data upstream.

Comparison with HTTP

Since RTP is rarely used without another protocol (such as HTTP) for non-transportation tasks, I will summarise RTP's strengths and weaknesses in the section on RTSP below.

3.4.4 RTSP

The Real Time Streaming Protocol (RTSP) [53] combines many of the characteristics of HTTP and RTP into a general, high-level streaming protocol suitable for on-demand and real-time applications, using either unicast or multicast distribution. In addition to merely acting as a source of streaming data, an RTSP server can also participate in conferences by recording or playing back content.

```
DESCRIBE rtsp://localhost/movie.mp4 RTSP/1.0
CSeq: 1
Accept: application/sdp
User-Agent: VLC Media Player (LIVE.COM)
```

Figure 3.4: Example of an RTSP DESCRIBE request. Note that request URIs are absolute. The CSeq field is incremented by one for each unique request.

RTSP uses very HTTP-like semantics and procedures for controlling streams (the similarity is intentional, and allows reuse of HTTP mechanisms such as authentication and security). In contrast to HTTP, streams are usually transferred separately from the control traffic. Also, while HTTP itself is a stateless protocol, state information is highly significant in RTSP. Finally, RTSP allows requests to be made both by clients and by servers (for instance to notify a client that a streaming parameter has changed).

In order to fulfill its role, RTSP supports streaming-specific functions such as time-based seeking and pausing of playback.

Requests

RTSP requests are structurally very similar to those defined by HTTP, and begin with a line containing a *method*, the URI it applies to and an RTSP version string; fields are separated by space characters (see figure 3.4). The first line is followed by zero or more header lines in exactly the same way as in HTTP.

Control traffic can run on top of both TCP and UDP (TCP is preferred). An RTSP server will therefore usually listen to requests on two ports (one TCP and one UDP). The default port number for both is 554.

A typical streaming session is initialised by a client sending a SETUP request for a stream or aggregate of streams (e.g. a TV feed comprised of one video and one audio stream). The request must contain the client's preferred transport mechanism (e.g. RTP unicast, see 3.4.3) as part of the header. If the requested resource is found (and the client is allowed to access it) the server prepares itself to start streaming, and returns an HTTP-like reply that includes the actual transport mechanism chosen[12], as well as an identifier that must accompany all further requests concerning the session.

When the session has been such readied, the client may instruct the server to start streaming by issuing a PLAY request.

RTSP includes several other methods; see [53].

Transport mechanisms

Delivery of streams is based on RTP, either on top of UDP (unicast or multicast) or TCP, or even embedded in the RTSP control channel itself.

RTSP is combined with multicast distribution either when an RTSP server provides descriptions for simple multicast sessions, or in the case of cyclical

[12]Any firewalls en route are expected to parse SETUP requests and make the necessary accomodations for streaming to work. It is therefore necessary that all such requests carry transport information, even in cases where the server dictates transport parameters.

on-demand applications (similar to traditional pay-TV systems with looping movies).

If firewalls or other network problems prevent pure RTP transportation from working, RTSP supports the embedding of binary RTP packets inside the connection otherwise used for control traffic alone. Because it increases overhead and complexity this should normally be avoided.

Advantages

Since RTSP and RTP are often used together, the advantages and disadvantages of both are combined in the following overviews.

RTSP and RTP have many positive qualities, mostly associated with their openness and being tailored for streaming applications. The most important are these:

- RTSP and RTP have been designed specifically for streaming. RTSP therefore supports useful features such as seeking, pausing and resuming.
- RTP streaming on top of UDP is more efficient than HTTP streaming, due to the limited overhead involved.
- Both RTP and RTSP are open standards, and a large number of implementations exist. Though not as ubiquitous as HTTP both are relatively well known among developers.
- Both protocols have been designed to be quite flexible and applicable to many problems.
- Obtaining perfect copies of RTP streams is not trivial, implying an obstacle for illegal copying.

Disadvantages

All is not rosy, though. Both protocols have problems, mostly associated with complexity, familiarity and the general problem of scalable streaming on the Internet:

- Both RTP and RTSP are considerably more complex than HTTP, making developing and troubleshooting applications that use them more difficult.
- Both protocols are less familiar to developers, administrators and users.
- A significant number of firewalls, NAT routers and other network equipment block or do not support RTP and RTSP properly.
- While RTP is more efficient than HTTP it still suffers from scalability problems. This applies even when ASM multicast is used, since clients are expected to provide RTCP feedback. (Additionally, ASM suffers from complexity, scalability and security problems itself[13].)

[13]Routing of ASM traffic for groups with many participants is challenging since it requires keeping very large routing tables. Also, any participant may transmit data, enabling both malevolent injection of data and denial of service attacks.

- As previously mentioned, making copies of RTP streams is usually relatively difficult. While this probably is a comfort to some content owners, it may represent a problem for viewers, listeners and users of streams.

3.4.5 SAP and SDP

It is sometimes desirable to announce multicast sessions without resorting to unicast mechanisms such as HTTP and RTSP. RFC 2974 [50] defines the Session Announcement Protocol (SAP) for distributing announcements to potential users via well-known multicast groups. (Both many-to-many conferences and one-to-many broadcasts are supported.)

SAP announcements are usually encoded according to the Session Description Protocol [54], which includes such details as session name; home page URI; contact information (email, phone etc.); and start and end times.

3.4.6 Proprietary protocols

Both Windows Media, QuickTime and Real seem to include proprietary streaming protocols, but all of them support streaming of their respective closed formats on top of RTSP and RTP.

Chapter 4

The Pycast encoding system

4.1 History

The multimedia project group at Østfold University College has worked with audio and video streaming since the mid-1990s. Early projects included streaming of the 1994 Olympic Winter Games at Lillehammer. This section describes the history of our MP3 streaming activities, which began in 1998.

In the beginning MP3 audio was encoded using simple shell scripts (implemented by Thomas Malt). Data was read either from a sound card or from a dedicated hardware MP3 encoder (the Netcoder, made by Audioactive [55]) via a serial port. The data would be sent directly to the streaming server, since it was already encoded. The raw (PCM) audio data from the sound card needed to be distributed to two or three encoder processes for encoding into different-quality MP3 streams (128, 56 and, optionally, 24 kbit/s). This splitting was achieved by sending the data to named UNIX FIFO pipes (one for each bitrate) using a series of —tee— instances. Each encoder process could then read the data from its assigned pipe, and send the encoded audio to the streaming server.

```
cd /tmp/

mkfifo nrk-p1-128 nrk-p1-56 nrk-p1-24 nrk-p2-128 nrk-p2-56 nrk-p2-24

sox /dev/dsp-p1 | tee nrk-p1-128 | tee nrk-p1-56 > nrk-p1-24 &
sox /dev/dsp-p2 | tee nrk-p2-128 | tee nrk-p2-56 > nrk-p2-24 &

lame -b 128 nrk-p1-128 | (echo 'header, nrk-p1-128...'; cat) | nc localhost 8000 &
lame -b 56 nrk-p1-56   | (echo 'header, nrk-p1-56...'; cat)  | nc localhost 8000 &
lame -b 24 nrk-p1-24   | (echo 'header, nrk-p1-24...'; cat)  | nc localhost 8000 &

lame -b 128 nrk-p2-128 | (echo 'header, nrk-p2-128...'; cat) | nc localhost 8000 &
lame -b 56 nrk-p2-56   | (echo 'header, nrk-p2-56...'; cat)  | nc localhost 8000 &
lame -b 24 nrk-p2-24   | (echo 'header, nrk-p2-24...'; cat)  | nc localhost 8000 &
```

Figure 4.1: Sketch of original FIFO-based encoding system

4.1.1 Problems

While the solution outlined above was appealing in its simplicity (for instance it consisted of only a small number of well-known and relatively user friendly UNIX commands) it had several disadvantages.

The script's most important shortcomings were instability and frequent errors at launch. Both were usually caused by buffer overruns that occured when one of the MP3 encoders was unable to read input data from its FIFO pipe rapidly enough (for instance if the encoder crashed). The overrun would cause one of the —tee— processes to terminate, thus breaking the entire command pipe, leading to a crash. The problem was especially common at start-up, since the delays associated with starting each process and an overall increase in CPU load increased the chances of overflowing FIFO buffers.

Also, while the system was conceptually quite elegant, its adherence to standard shell mechanisms like pipes, redirects and spawning of background processes (each branch or sub-branch of the pipe hierarchy needed to run in parallel) led to very verbose and redundant code. For instance adding a new bitrate involved splicing the associated code (which was almost, but not completely, identical to the code for the other bitrates) into an already very long command line. The result was that changes, debugging and improvements were difficult to perform, discouraging experimentation.

4.2 Pycast

It was clear that the system needed fundamental changes to keep up as the number of channels, bitrates and distribution methods grew. It was also quite obvious that configuration information should be moved to a separate file, and not be interspersed with the programme code.

One option was to keep using shell scripting, but organise the code using more traditional programming constructs. Unfortunately shell code has a tendency to lose much of its elegance when that threshold is crossed. A prototype was therefore implemented in Python, and early tests confirmed that we were on the right track. (The Python code was much clearer, and everything indicated that it would scale well.)

Much of the original system's pipeline metaphor was retained in the Python version. Speficically, the significant command line components (e.g. data sources, encoders and streaming server transmitters) now appeared as classes, and the component structure was specified in a hierachically origanised XML file. The application was (not surprisingly) named Pycast.

We have now used Pycast for MP3 and Ogg continously since the summer of 2001. Among other things it has been upgraded with functionality for encoding more audio formats, multicasting, and clustering.

4.2.1 Physical structure

Our real-time radio streaming system, shown in figure 4.2 currently[1] consists of four encoding servers running Pycast, feeding an Icecast streaming server.

[1] May 2005

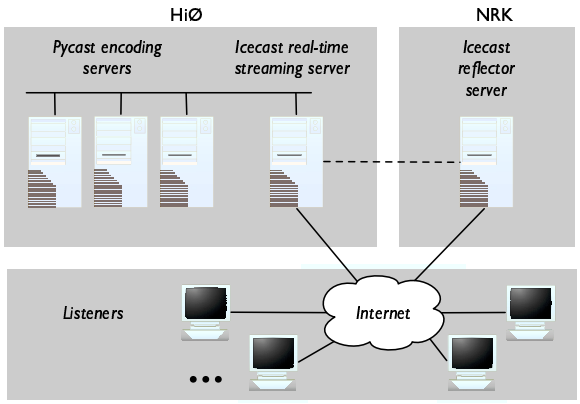

Figure 4.2: Physical structure of HiØ's real-time radio streaming system

A second Icecast server located at NRK's headquarters in Oslo, acting as a reflector, redistributes all of the streams, reducing the load on HiØ's Internet connection. (The simple load management scheme used works by generating M3U files[2] that direct half of the listeners directly to HiØ's streaming server, and the rest to the reflector.)

Three of the encoding servers are part of an openMosix [56] cluster, and receive a Digital Video Broadcasting (DVB) transport stream (TS) for transcoding from the Thor 3 satellite. The fourth server (not shown) is used to transcode the NRK P1 channel as received via Digital Audio Broadcast (DAB)[3].

The streaming server is also responsible for distributing all of the radio channels via IP multicast.

4.2.2 Overview

Pycast integrates the various components involved in capturing, digitising, encoding, transcoding and distributing audio streams. The source of a stream is usually a capture card (e.g. a DVB card or analogue sound card), but receiving data from a streaming server is also supported. Digitisation using suitable parameters (sampling rate, resolution etc.) is needed when reading from analogue sources. Pycast uses tools from the LinuxTV project [57] for DVB input and Sox [58] for analogue sampling and processing.

At the moment Pycast can encode to MP3 (using Lame [59]) and Ogg Vorbis (using OggEnc [60]), but it is quite simple to add support for other formats. Streams can be distributed via Icecast [1] 1 (MP3) and Icecast 2 (MP3 and Ogg Vorbis) streaming servers, as well as RTP multicast (MP3).

A typical structure for streaming one radio channel in several bitrates and formats is shown in figure 4.3.

[2]M3U files are used to direct media players to audio streams, and usually consist of one single line of plain text with the URL of the stream in question. Such files are necessary to ensure that streams are not opened by web browsers themselves.

[3]The DVB version of NRK P1 contains regional programming that is not always in sync with the national radio schedule. Using the DAB feed therefore eliminates a potential source of confusion.

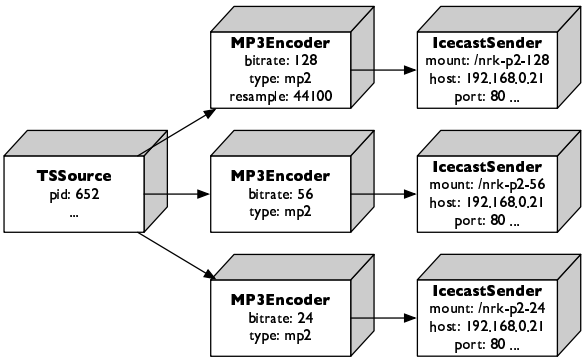

Figure 4.3: Typical Pycast structure. The output from a TSSource object (an MP2 elementary stream) is split and flows to three MP3Encoder objects. Each MP3Encoder object handles transcoding of a stream from MP2 to MP3, based on a set of parameters (e.g. bitrate). The resulting MP3 streams are forwarded to an Icecast server via corresponding IcecastSender objects, placing each stream at its appropriate URI (mountpoint) on the server.

Here, an unencrypted MP2 elementary stream is extracted from a DVB transport stream. It is then cloned and sent to six encoder objects to be transcoded to MP3 and Ogg Vorbis at different qualities. (The MP3Encoder module can optionally transcode streams by decoding them first. This might seem less efficient than using one common object for decoding, but in its current form Pycast works better with short chains of objects.)

Each encoded stream is then fed to the streaming server via its own IcecastSender. (RTPSender objects could also be added alongside the IcecastSenders to enable multicast streaming.)

See figure 1.1 for an overview of how Pycast is integrated with other applications.

4.2.3 Design method

Pycast began as a simple application to replace tools that did not perform satisfactorily (see 4.1.1). It has since been extended and altered to solve problems.

In other words Pycast as a system has grown quite organically. As a consequence it suffers from a lack of consistency. This should be corrected in future work.

4.2.4 Modules

As indicated in the figures, Pycast is based on modules processing data in a tree-like structure. Modules are implemented as classes, and inherit from one or both of the superclasses Producer and Consumer, which are both children of PipelineComponent (figure 4.4).

PipelineComponent implements basic functionality common to all modules. Producer objects generate data, either by processing data from another

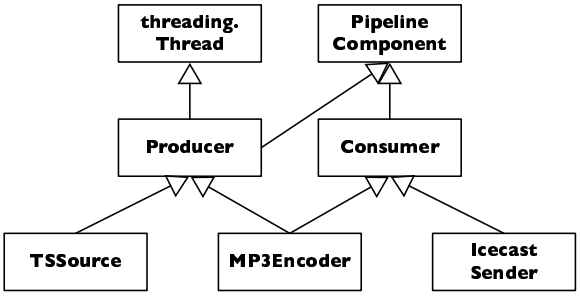

Figure 4.4: UML diagram of representative classes

component (e.g. MP3Encoder) or by importing data from an external source (e.g. TSSource), and sending it to one or more Consumer objects. Consumer objects receive data from a Producer and either process it for distribution further down the pipeline tree (e.g. MP3Encoder) or send it somewhere outside the Pycast system (e.g. IcecastSender). Most classes are both Producers and Consumers.

Because Producers need to operate asynchronously, the Producer class also inherits from Thread in Python's threading module.

Common modules

Pycast has a number of modules for various purposes, however most of them are experimental. The most commonly used (and thus, most thoroughly tested) are these:

HTTPSource Reads data from an HTTP streaming server. Coupled with an RTPSender it is useful for relaying a stream for multicast distribution.

IcecastSender Sends a stream to an Icecast-compatible streaming server (older versions of Pycast supported Icecast 1 servers; the current version supports Icecast 2).

MP3Encoder Encodes its input data (either raw audio or MP2/MP3) into MP3 using Lame.

RTPSender Multicasts its input data to a specified group using RTP. For partially unknown reasons the current implementation is incompatible with QuickTime Player and RealPlayer, but works well with VLC.

SoundcardSource Reads data from an analogue sound card using SoX [58].

TSSource Receives an MPEG transport stream multicast by dvbstream [61] and extracts a specified elementary stream. (dvbstream runs separately, often on another computer.)

```
<pycastconfig>
  <tssource group="224.55.44.30" port="5000" pid="652">
    <mp3encoder bitrate="128" type="mp2" resample="44100">
      <icecastsender mountpoint="/nrk-p2-128" password="xxx"
        bitrate="128" header="header-nrk-p2-txt"
        host="192.168.0.21" port="80" />
    </mp3encoder>
    <mp3encoder bitrate="56" type="mp2">
      <icecastsender mountpoint="/nrk-p2-56" password="xxx"
      bitrate="56" header="header-nrk-p2-txt"
      host="192.168.0.21" port="80" />
    </mp3encoder>
    <mp3encoder bitrate="24" type="mp2">
      <icecastsender mountpoint="/nrk-p2-24" password="xxx"
      bitrate="24" header="header-nrk-p2-txt"
      host="192.168.0.21" port="80" />
    </mp3encoder>
  </tssource>
</pycastconfig>
```

Figure 4.5: Typical configuration file

VorbisEncoder Encodes its input data (only raw audio is supported) into Ogg Vorbis, using OggEnc.

Configuration

As previously mentioned Pycast is configured using an XML file, since XML's hierarchical structure maps well to the internal module structure. An example is shown in figure 4.5 (the file describes the structure in figure 4.3).

4.2.5 Parallel processing

Since Pycast consists of components that pass relatively low-bitrate data[4] between each other in quite predictable ways it lends itself well to parallelisation.

Parallel processing with Pycast is useful when encoding more streams than one single computer can handle. We initially distributed the encoding workload manually between our servers, by running a separate Pycast instance on each. The main advantages of such an approach are simplicity and separation of work, so that if one computer goes offline it will only affect a subset of the streams. On the other hand it makes it more difficult to efficiently exploit the available CPU power, both because distribution by hand is more coarse-grained (only entire channels can easily be moved) and because it can lead to unneccessarily complex configuration files.

To make parallelisation simpler and more flexible we currently use openMosix [56] for automatic migration of processes in a four-node cluster. openMosix seems to handle our needs very well, and it is very easy to add or remove servers from the cluster: Slave nodes are booted using DHCP [62] and TFTP[63], and as long as there is spare capacity in the cluster processes will migrate away from slave computers that are gracefully shut down, with no

[4]The worst case is transmission of 48 kHz raw, stereo audio, with a data rate of 1.5 Mbit/s. Even on a cluster with a 100 Mbit/s internal network this is unproblematic.

consequences for encoding quality or stability. If a slave node crashes the processes it contains will be lost; Pycast reacts to this as if the missing processes have crashed and starts corresponding new ones after a small delay.

If a cluster (or, indeed, one single computer) reaches more than full capacity it will not be able to encode audio quickly enough. In the better case this leads clicks and pops in the aduio stream; in the worse case components will start to crash because of buffer overruns. We therefore keep an extra node online in case one of the others should fail.

Using openMosix has worked well, reducing administrative load and making it easy to add or remove nodes without temporarily stopping one or more channels. However, since all encoding now relies on the master node the system as a whole has become more susceptible to single-point failures, and it has also become more difficult to debug.

4.2.6 Performance

We have used Pycast for all radio encoding since the summer of 2001, and it has mostly worked very well. It should be noted that our realtime radio streams are formally considered experimental and are operated with limited resources. For instance, changes are usually incorporated into the production system after relatively little testing.

CPU and memory efficiency is more than acceptable; the Pycast code itself (excluding external programmes) usually consumes less than 2% of both. The automatic openMosix migration usually works well.

Problems do occur from time to time, the most common being these:

- Satellite reception problems can cause dvbstream to crash, taking all DVB-based streams down with it. This used to be a common problem, but was solved by aligning our satellite dishes more carefully.

- An important design goal has been to isolate all Pycast components from errors in upstream objects (i.e. receivers of data) or those in other branches. For reasons that we have not yet had the opportunity to look into, the isolation mechanisms currently do not work as intended. For instance, problems on a streaming server can cause the entire application to slow down and lose data; ideally only the object sending data to the server should be affected.

- Components that rely on an external process currently use a relatively unsophisticated method to check if it has crashed: If no data has been received from the process within a certain amount of time, something is assumed to be wrong and the external utility is restarted. During periods of high system load, for instance when Pycast is started, this mechanism can temporarily make things worse by adding the overhead of process startups and terminations. (This problem is most of all just annoying, and usually does not stop the system from working.)

- Pycast sometimes crashes or needs to be restarted for no obvious reason. Problems like these typically occur around once per month, and can almost always be fixed by completely stopping and restarting it.

4.2.7 Further development

In the short term the issues most important to address are those described above. In a slightly longer perspective we would like to add or change more interesting features, such as the following:

Enhanced DVB and DAB integration

Pycast's DVB support is currently a bit of a hack, based on receiving and demultiplexing an MPEG-2 transport stream multicast by the dvbstream utility (see 4.2.4). This works relatively well, but it might be better to interface directly with the DVB card drivers using library calls. Most importantly it would give Pycast better control of its sources, and make troubleshooting and debugging easier. (Since it is not a CPU-intensive activity, little potential for parallelisation is lost by not using an external process.)

It would also be useful to add support for receiving DAB streams directly from a PCI card or an external tuner. This would have many of the same advantages as using DVB (e.g. being able to receive several multiplexed streams simultaneously), and also allow reception of streams not available via DVB (such as regional channels).

Administration

It is sometimes desirable to be able to treat data branches individually, for instance when a specific stream or channel needs to be restarted with a configuration data. With today's version of Pycast, however, it is necessary to restart the whole application whenever even a small change has been made to its configuration file. Making the Pycast administration interface more fine-grained would therefore be quite helpful.

Other improvements

Pycast would also benefit from monitoring support (e.g. displaying detailed status information via a web interface); support for other source types, codecs and streaming servers; as well as integration with more external tools and libraries.

4.3 Statistics

We have systematically collected listener statistics for our unicast radio streams since 17 September 2001; every five minutes the number of listeners connected to a stream is measured via Icecast's administration interface and recorded.

Figure 4.6 shows the highest number of listeners each month between September 2001 and early May 2005.

Listening patterns for a typical week can be seen in figure 4.7, which shows the number of concurrently connected listeners for a period of seven days (sampled every five minutes). It also indicates when network problems or other unexpected events occurred, manifesting themselves as sudden drops or spikes.

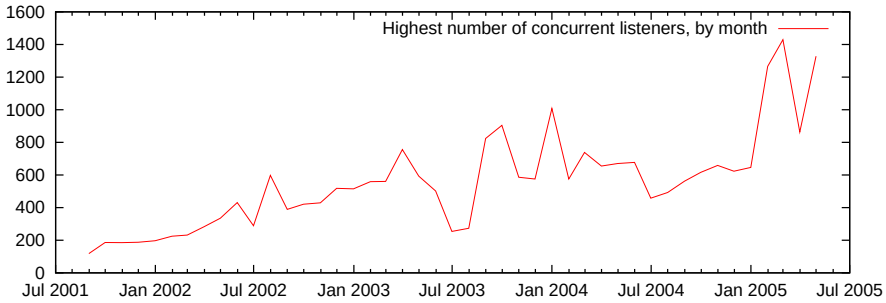

Figure 4.6: Monthly listener maxima since September 2001. The graph shows the highest number of concurrently connected listeners each month. Notice the annual fluctuations, with low points every summer.

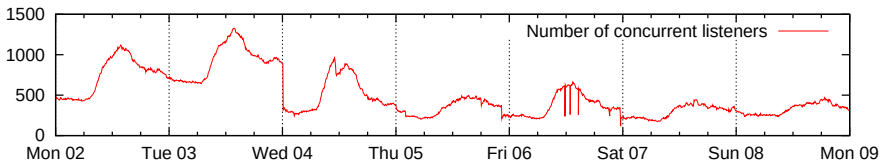

Figure 4.7: Concurrent listeners during first week of May 2005. The total number of currently connected listeners is measured every five minutes. Notice the daily fluctuations, and the reduced number of listeners during the weekend (as well as Thursday, which was a holiday). Sudden value changes are usually caused by network problems.

Chapter 5

The Tista streaming server

5.1 History

5.1.1 Radio on demand, version 1

Our first radio on demand (RoD) system was developed by Torbjørn Mathisen and John-Olav Hoddevik during the spring of 2000, as part of a third-year student project [64]. It automatically recorded one of NRK's radio channels[1] (NRK Petre), and the recorded programmes were made publically available on the web. The service became quite popular, with several hundred downloads per day, and was therefore kept running after the end of the project.

Our ambition was to store programmes indefinitely, and programmes were never intentionally erased. However, since the system ran on low-cost hardware and with limited resources, data was sometimes lost, mostly because of failing hard disks. Backups were not performed.

Hardware

The system was distributed across three identical PC-based servers (figure 5.1) running Linux. One server (henceforth called the *primary server*) handled recording and application logic; the others were used to store and serve MP3 files. All of the servers were connected to the Internet; clients would stream a programme directly from the server on which it was stored, instead of streaming via the primary one.

Storage capacity was later increased, either by adding dedicated computers, or by using available space on other servers. Remote disks were mounted via NFS, and file serving was moved to the primary server for improved security and easier management. NRK covered hardware costs (approximately NOK 20,000 per year), since the project had attracted many listeners and yielded interesting insight into on-demand streaming.

[1] NRK broadcasts nine national radio channels, of which P1 (light entertainment), P2 (special interest) and Petre (youth and young adult) are the most popular.

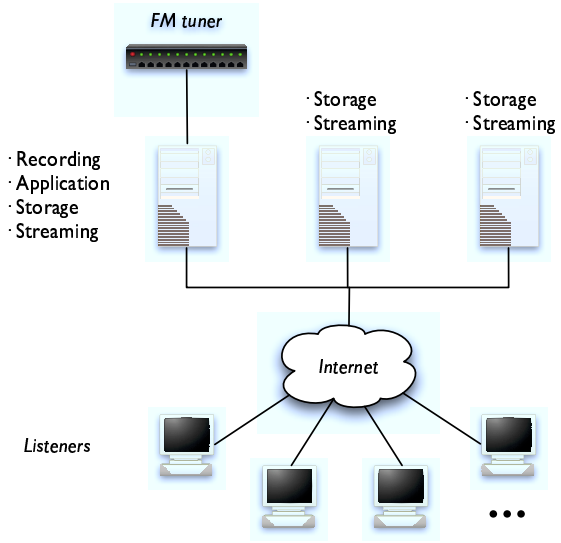

Figure 5.1: The original radio on demand system

Software

The system was simple but effective: A script ran on the primary server once every night (via `cron`[2]) and downloaded the next day's programme schedule from NRK's website. The retrieved HTML file was parsed, and the programme metadata (including start time, title and programme description) stored in a MySQL [65] database. A series of `at`[3] jobs were then created to start and stop the recording of each programme.

All updates to the programme database were made during a six-hour window each night when no programmes were broadcast and recording was turned off.

Audio was initally sampled and encoded locally, but to make management simpler, the recording subssystem was later modified slightly to use already encoded MP3 streams from our unicast radio server instead.

The hard disks of the two secondary servers were mounted on the primary server using NFS. The disk used for storing new files was selected cyclically at the beginning of the day. This worked well in the beginning when all disks were of identical size, but the script was later modified to choose the disk with the most available space.

5.1.2 Radio on demand, version 2

In the early spring of 2002 we were asked by NRK to extend the radio on demand system to record two other channels (NRK P1 and NRK P2) in addition

[2]The UNIX subsystem for periodic execution of jobs.
[3]The UNIX subsystem for timed (but not repeating) execution of jobs.

to NRK Petre. Programmes from the two new channels would be used exclusively in-house by NRK for planning, and not be made available to the public.

To accommodate the new requirements several changes needed to be made to the system, including adding support for round-the-clock recording (NRK P1 and NRK P2 are broadcast continuously) as well as improvements to metadata handling and the web search interface. Additionally, much of the code was refactored, and downloading of programme schedules made more robust by retrieving information several days in advance (thus, if a schedule failed to download one night it would probably work the nest). Most of the changes were implemented by Håvard Rast Blok.

We continued to add storage capacity by purchasing dedicated (but low-cost) NFS servers, and by using surplus space on other computers.

It became increasingly obvious that such a simplistic solution led to complex problems when scaled up: Recording three channels, each at 128, 56 and 24 kbit/s, consumes six gigabytes of disk space per day. As the number of hard disks and servers grew, hardware failures became frustratingly common.

5.1.3 Lessons learned

The experiences from developing and running versions one and two of our radio on demand taught us useful things:

- Successful recording should not depend on having a correct schedule, or having a schedule at all. In our system a missing schedule meant that recording would not be started, and schedule errors that recording would start or stop at the wrong times. Such errors therefore led to problems that were respectively either impossible or difficult to correct.

- Since a standard web server (Apache [66]) was used for streaming the recorded files it was difficult to extract arbitrary parts of a programme. Clients that use the HTTP Range header [48] (e.g. WinAmp) do allow a user to move around quite freely inside a streamed file using educated guesses based on the file's bitrate. (In a file with a well-known and constant bitrate the relationship between the time offset t and the byte position is simply $bt/8$, where b is the bitrate.) Unfortunately, there is no standardised way of indicating the desired offset in a file as part of a URI.

- A multi-computer system whose well-being depends on all its participating machines to be working well can become very fragile, unless it incorporates some amount of redundancy. In our case adding such redundancy was not trivial.

- Hard disks and other hardware regularly fail. In systems with many computers failures can become very common.

- Cheap hardware fails more frequently than server-grade hardware, but is often easier (and less expensive) to replace or repair.

- Simple solutions do not necessarily scale well.

5.2 The Tista streaming server

In early 2003 NRK decided to officially incorporate radio on demand as a part of their official Internet services. We were therefore asked to improve our radio on demand system so that it could be used in a relatively large-scale production environment. The result was the Tista[4] streaming server.

5.2.1 Requirements

In co-operation with NRK we identified the following key requirements:

- It would be integrated with NRK's existing services and infrastructure, and should therefore be easily manageable and at least as robust as other similar systems at NRK.
- It should be based on MP3 audio streamed over HTTP, for maximum client compatibility.
- It should scale to a large number of simultaneous listeners (the original goal was 1000, later reduced to 600, due to hardware limitations).
- It should be able to store at least three weeks of audio data for three channels, at three bitrates (128, 56 and 24 kbits/s).
- Storage should be very reliable.
- It should work on Intel-based hardware running Red Hat Linux.
- After development had finished, it should be easy to move the system to NRK and put it into production.

5.2.2 Design method

The design process was iterative, and consisted of the following stages:

- Identifying system requirements (see 5.2.1).
- Identifying and avoiding the most serious shortcomings of the previous versions (see 5.1.3).
- Quickly developing and testing simple prototypes (see 5.2.3), until we were reasonably we had found one worth elaborating on.
- Adding functionality incrementally, and testing it reasonably thoroughly for each step. (We did not, however, use automatic code testing of any sort; that is worth considering for further work.)
- Testing the final system thoroughly, including subjecting it to realistic loads and usage patterns (see 5.3.3). If problems were discovered points 4 and 5 were repeated, until the system was deemed fit for deployment.

[4]For the curious: The server is named after the river Tista which flows through the town of Halden, where we are located.

5.2.3 Design choices

The project offered the opportunity to correct well-known flaws and shortcomings in the older RoD versions. Few serious problems were encountered during the implementation[5].

The most important design choices made are outlined here.

Programming language

The two candidate implementation languages (because of our previous experience with both) were Python and C (C would have been used to write light-weight CGI scripts that could be used in conjunction with Apache).

The initial plan was to base the streaming server on Apache, and use mod_python [67] or light-weight C programmes to implement support for specifying arbitrary time intervals (start and stop times) as part of a URI.

Early tests showed that the overhead associated with such a solution was too high, since Apache allocates a separate thread (or process) to each request. Even serving files directly with Apache (which the original RoD system did) scales poorly. (Apache's scalability problems are not too surprising in retrospect: It is designed for serving relatively small files, not for continous streaming).

Other, more light-weight web servers were not considered. This was an oversight: If a suitable server had been found and proved to work well, it could have saved us some time, but we believe that the end result would not have been significantly better.

Python turned out to be a good choice. I will refrain from elaborating this conclusion in detail, and limit myself to mentioning these few points:

- The interactive interpreter aids debugging and experimentation.
- Solving problems in Python requires fewer lines of code than languages such as Java, C and C++. Less code equals fewer errors, and makes reading and understanding the code easier and quicker. Also, a larger portion of a programme can be kept in the brain's short-term memory.
- The lack of a separate compilation stage leads to less time wasted waiting for results.
- The stricter code layout rules (compared to Java, C and C++) aid comprehension.
- The built-in types and data structures reduce the time wasted implementing such constructs, or working around language shortcomings.
- Python, unless typical scripting languages like Perl, scales well even for large systems.

[5] One unfortunate surprise late in the project was that one common version Windows Media Player had signficant problems streaming MP3 audio. This problem should be considered a result of the requirements specification.

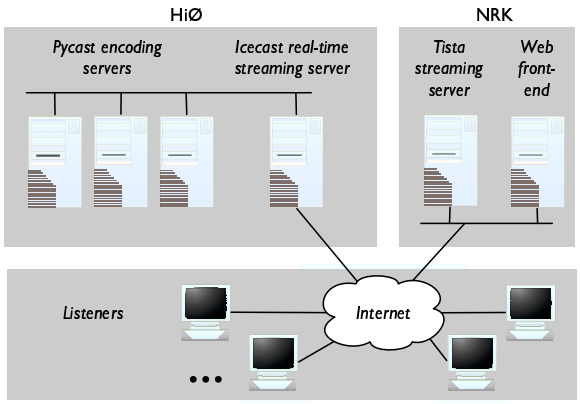

Figure 5.2: Physical structure of NRK's radio-on-demand system

Schedule sensitivity

It was clear early in the design process that the recording susbsystem should not have to depend on correct programme scheduling information.

Storage

The most important requirements for the storage medium was scalability (to a large number of concurrent users), redundancy and reliability.

An ATAboy IDE-based RAID array (with a SCSI interface) from Nexsan Technologies [68] was chosen. It supported RAID 5, had plenty of space, relatively high performance, and was based on low-cost IDE disks.

Metadata storage

Metadata need to be stored to enable searching, and so that information such as programme names can be displayed in a user's media player during playback. Tista had to be able to store, query and retrieve a relatively large number of programmes[6], making it impractical to use regular filesystem storage.

MySQL was chosen, since it was already in use at NRK, and is easily integrated with Python.

5.2.4 Physical structure

Figure 5.2 shows the hardware configuration that was used to run NRK's on-demand system. Computers at HiØ encoded the various streams, which were then distributed via an Icecast server.

The Tista recording server, located at NRK's headquarters in Oslo, received the MP3 streams via the Internet, in the same way that any other client would.

[6] Assuming an average of two programmes broadcast per channel per hour for three weeks, it is necessary to store 1008 records to cover a period of three weeks, or more than 3000 for three channels.

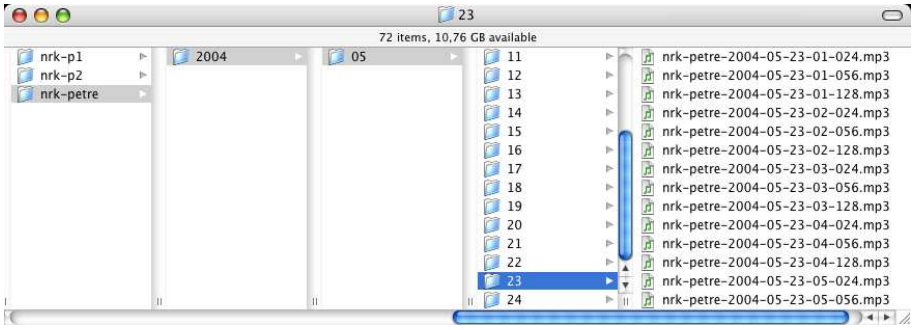

Figure 5.3: Directory structure

The user interface was generated by another web server at the NRK, serving as a user-friendly front end to the system.

5.2.5 Recording

To avoid the problem of programme schedule dependence, recording is done in one-hour *chunks*[7]. Each stream is continuously stored on disk. At the start of a new hour a new file is created to store the next hour's worth of data. Files are named based on a stream's name, its format and bitrate, as well as the date and hour that the file was created (figure 5.3). A file is therefore easily identifiable later even if it has been moved from its original directory.

Files are organised in hierarchical directory trees, potentially distributed across several disks or partitions. If more than one mountpoint is used new files will be placed where the most space is available.

Tista currently assumes that all streams have constant bitrates (CBR). (It can easily be extended to support variable bitrate (VBR) data as well. However, calculating byte offsets in VBR files is slightly more difficult, and has not been implemented yet.)

The recording server

The recorder (`record.py`) runs in the background as a daemon, and continuously downloads and stores on disk one or more MP3 streams from an HTTP-based streaming server (see figure 1.1).

Each stream is handled by its own thread. If the connection to a source is lost, the thread will wait for a configurable number of seconds (default: 30) before trying to reconnect. (The wait should be short enough to cause minimal loss of data, but long enough to avoid unduly stressing the streaming server or the network.)

The recording daemon also contains experimental support for archiving Windows Media (WMA) streams, based on the freely available mmsclient[69] utility. Recording does work, but archived WMA streams can not yet be streamed

[7] The choice of one-hour chunks was quite arbitrary. One advantage is that the number of files stored in each directory is quite low. A distinct disadvantage of such a large chunk size is an increased risk of missing programmes

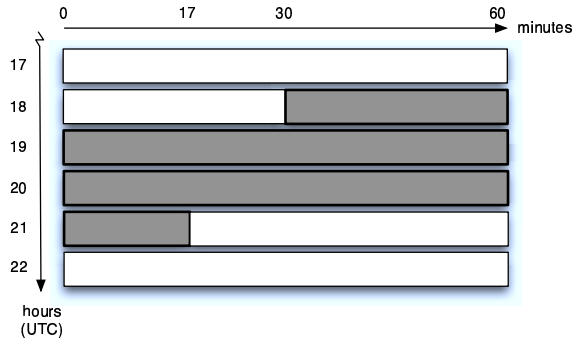

Figure 5.4: Reassembling a programme (in this case one that lasts from 18:30 to 21:17 UTC)

to clients. (Most WMA clients do not support reading from arbitrary points in a stream. Full WMA support therefore requires a seeking algorithm that is aware of a WMA file's structure.)

5.2.6 Reassembly

Programmes are assembled by the streaming server from one or more files on demand (figure 5.4). If a programme starts after the exact beginning of an hour a seek is performed to the corresponding point in the first file (based on its bitrate) before data is read. The amount of data that is read is adjusted similarily if the programme ends before the end of the last file that contains a part of it.

When the streaming server receives a request it will check that the requested interval is shorter than a configurable limit (default: 12 hours), to discourage abuse of the service (i.e. downloading extremely large portions of archived streams).

Streams are stored and reassembled in such a way that no data is lost between chunks. (Chunks are recorded "back-to-back", and no data is ever discarded between the closing of one chunk storage file and the opening of the one that follows.)

5.2.7 The streaming server

The streaming server is Tista's most important component, handling the extraction and streaming of archived radio programmes to a potentially large number of simultaneous listeners. It uses HTTP/1.1 [48] as its transport protocol, and is therefore compatible with the vast majority of media players.

Implementing a server capable of handling hundreds of simultaneous, long-lived clients required a different approach than what is often sufficient to solve simpler problems.

The main difference between Tista and most common web servers is the time spent handling a typical request: the longer it takes to finish sending a reply, the higher the number of active connections on the server.

Most web sites consist of mostly relatively small files (e.g. HTML files, images and other documents), which are downloaded by clients at maximum speed. In most cases a request can therefore be received, processed and a reply completed in just a few seconds or less. Even if such a server receives high number of requests per hour, the number of active connections at any point in is typically quite low.

Streaming is significantly different for two reasons: The amount of data transferred is usually large than for typical file requests (e.g. one hour of 128 kbit/s MP3 audio is more than 60 MB of data), and most clients will download the data at the same rate at which it is played back. The majority of connections will therefore last for several minutes or even hours, and the number of active connections at any time will typically be high. This implies a large number of open sockets and files.

A client's download rate is limited to the requested programme's bitrate multiplied by a configurable factor (default: two). This is done to keep load at a minimum (thus improving the quality of service for other clients), and to discourage mass-downloading of the server's contents.

As long as Tista is to be used for real-time streaming of on-demand content, the transfer rate factor should be kept well above one, to ensure the clients' pre-buffering phase (the time that a media player spends buffering data before playback actually starts) be as short as possible.

Tista uses Psyco [70] for just-in-time compiling, if it is installed. All performance testing of the streaming server was done with Psyco enabled, and Psyco was also used while the system was in service at NRK.

Threads vs. `select`/`poll`

One obvious way to organise the serving of data to many clients at the same time is to assign each client a thread (or process) responsible for communicating with it. Unfortunately, such a solution does not necessarily scale well. Experiments, with both Apache and a stand-alone Python-based server, indicated that heavy multithreading carried with it a significant overhead (CPU-wise and probably also memory-wise) that drastically limited the number of simultaneous clients. Spawning separate processes would necessarily be even more demanding for the computer.

A different, and in our case better, approach is to use the operating system's `select` call (interfaced by Python's `select` module [71]) to periodically check which clients are ready to receive data. This makes it possible to efficiently handle at least several hundred clients at a time in a single thread. Tista still uses several threads (e.g. one for listening to new requests, and one for streaming), but using `select` keeps the number low.

By using Python's `poll` class [71] (also an interface to underlying OS functionality) it was possible to increase performance even more by avoiding inefficiencies in `select`.

One disadvantage of using Python is that threads are tied to the interpreter process, and can not be independently moved to other CPUs in a multiprocessor system.

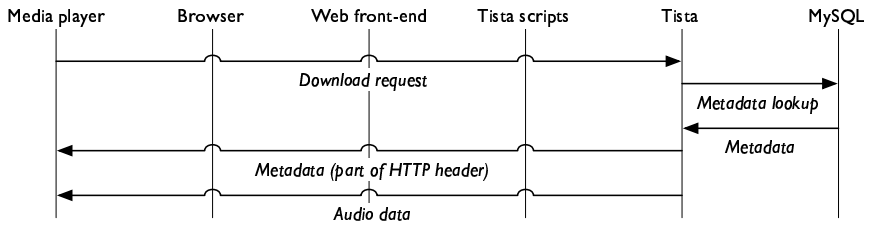

Figure 5.5: Download sequence. Web front end *is the web server running NRK's user interface.*

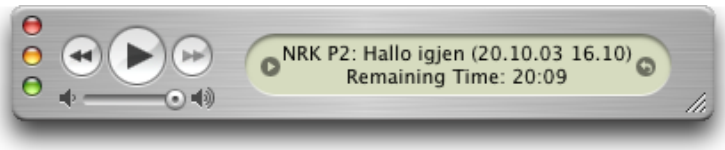

Figure 5.6: Example of metadata displayed during playback: Channel name followed by programme name, and the date and time it was originally broadcast. The media player (in this case Apple's iTunes [72]) calculates the programme's length based on its bitrate and size.

File descriptors

Linux uses file descriptors to represent both open files and sockets (a file usually needs one descriptor; a bi-directional socket needs two). Tista must therefore allocate three file descriptors for every listener (two for the socket and one for the file object reading data from the disk).

In other words Tista potentially needs significantly more file descriptors than a typical web server. Most operating systems limit the number of file descriptors available to processes run by regular users (it is recommended that Tista be run by a regular user, not root). It is therefore advisable to raise the operating system's FD limit (this is done automatically by the included init script).

Metadata handling

At startup the streaming server will try to connect to the metadata database (see 5.2.8), and, if successful, use the connection to look up metadata (e.g. programme names) for incoming requests. Metadata are sent as part of the HTTP header (figure 5.5), so that media players can display relevant information during playback (figure 5.6).

Synchronising MP3 streams

When starting a new MP3 stream (i.e. sending the first audio data to a new listener) the server will always start at a frame boundary (by scanning forwards

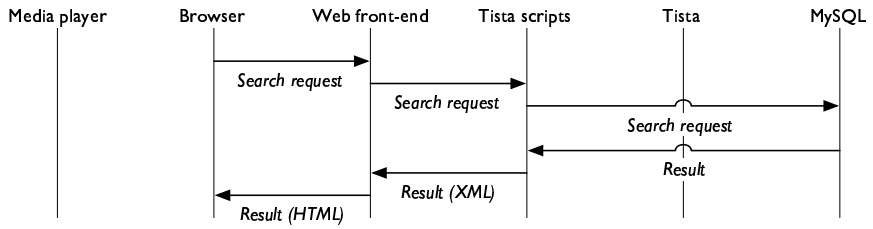

Figure 5.7: Programme search sequence. The search script returns an XML file representing the results, which NRK's front end web server presents using HTML.

in the data stream until one is found) to accommodate media players that expect this.

Logging

The server maintains two logs, one (`tista_log`) in a custom format (see Appendix F), and one (`access_log`) in Apache's default access log format. The former can log more details (detail level is configurable)[8]; the latter is compatible with existing log analysis software.

5.2.8 CGI and mod_python scripts

Tista includes a number of external CGI and mod_python scripts to handle metadata searching, M3U file generation, and receiving and storting schedule information. These functions could have been integrated into the streaming server itself, but we chose not to, since Apache already has excellent scripting support.

The CGI and mod_python scripts are designed to be easily integrated with Apache 2.0.

Searching and M3U generation

Since searching and generation of M3U files are very common operations (a listener will typically do both for every programme retrieved) these functions are implemented using mod_python for minimal overhead[9]. Both are defined in the same file but presented on the Web at different URIs using Apache's *RewriteEngine* [73].

The search script was designed to work as an interface between Tista and NRK's own search scripts. Given a channel name, bitrate, format, start time, end time, and optionally a parameter to limit the number of hits it will return a list of zero or more (always lower than a hard-coded maximum value and

[8]For instance, Apache only creates log entries for finished requests. This makes sense for short requests, but not for streaming requests which can take minutes or hours to complete.

[9]With mod_python Apache does not need to start a new Python interpreter for every request. Also, resources like database connections can be reused.

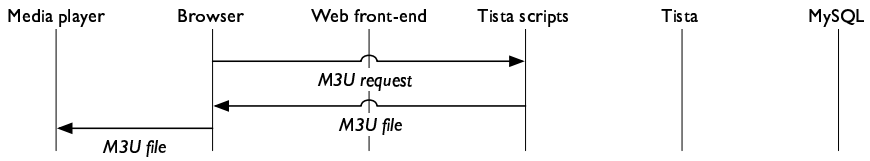

Figure 5.8: M3U generation sequence. The web server sends the M3U file to an appropriate media player.

possibly also limited by the maximum hits parameter) URIs for matching programmes (figure 5.7). The results are structured using a custom XML format (see Appendix E), but if the optional parameter `html=1` is supplied, they will be presented using simple HTML. (This parameter is only intended to be used for testing and debugging.)

The search script can be found at `/services/public/search` on the web server (figure 5.8).

A second script (also implemented using mod_python) receives the same set of parameters that the search script returns, and produces a simple one-line M3U file that refers to the actual programme, i.e. a URI on the Tista streaming server.

Metadata reception

A CGI script (`/services/private/gluon_receive`) is used to receive, parse and store programme information. Tista was designed for easy integration with NRK's existing metadata distribution system, Gluon, which pushes to Dublin Core [74]-based metadata files to interested clients. Gluon is used to transmit many different kinds of information in addition to programme schedules (e.g. sports scores), but clients can choose to be sent only specific categories of data (in this case programme information).

When new programme data is available it will be fed to Tista's `gluon_receive` using an HTTP POST request. It is then parsed and stored in the MySQL database (using channel ID and start time as the primary key, see C). Information concerning channels that should not be archived is ignored. (Gluon distributes metadata for several radio and TV channels.)

Monitoring

Tista also includes a CGI script for simple monitoring of the server (`/services/private/status`), which displays the server's uptime and load, interesting processes, number of recorders running, number of listeners and the last three lines of the main log file (`tista_log`, see Appendix F).

The monitoring script is useful for administrators to check if Tista is working properly. Since a relatively small amount of information is displayed, it even works well with cell phone browsers.

5.2.9 `cron` scripts

Tista includes two `cron` scripts: one for automatically deleting expired audio files (thus conserving disk space), and one to maintain a table in the database of available programmes.

The file deletion script (`cron/delete_old.py`) is run nightly to delete audio files older than a specified limit, as well as all directories that no longer contains files or directories.

The search script needs to know which programmes are actually available for download: A programme will be unavailable if it has not yet been recorded, if sigificant errors occured during recording, or if it has later been deleted.

Because availability checking is a relatively costly procedure that has to be performed for every programme within the scope of a search query, and because it can not easily be integrated with SQL code, availability status is cached in the database (see Appendix C). It is peridocally updated by the `update_availability.py` script, which checks the status of recent programmes every five minutes, and checks older programmes every 24 hours.

5.2.10 Common library

The library `rod3.py` (so named for historical reasons) is shared by Tista's other components, and implements and defines various functions and constants[10].

5.2.11 Times and dates

To avoid confusion and improve consistency, all times used for storing and retrieving files are in UTC and formatted according to ISO 8601 [75] as described in RFC 3339[76]. This means that the system can easily handle transitions between standard and daylight saving time. It also simplifies management of a RoD service that is used across time zones.

Conversion between UTC and other time zones is handled by other parts of the system (e.g. the web interface).

To ensure consistency, all involved computers should use NTP [77] for time synchronisation.

5.2.12 Error tolerance

As previously mentioned, Tista assumes that the bitrate of all streams is constant and precisely known, as all extraction operations depend on it. The files stored on disk are raw dumps of the data received.

This approach has two important disadvantages: It can not easily be used to store variable-bitrate streams, and if errors occured during recording (or streaming) entire one-hour chunks of data could be rendered useless. (Even a small error will lead to lost data, and thus influence all offset calculations.)

Tista can therefore be configured to consider a file to be lost if its size differs more than a certain percentage (default: 4%) from its expected size.

[10] Strictly speaking Python does not have constants. The term is used here to indicate a variable that should not be changed after its initial definition.

```
http://malxrod01.nrk.no:8000/archive?channel=
nrk-petre&bitrate=128&start=2005-04-03T13:03:
00&end=2005-04-03T14:00:10
```

Figure 5.9: A typical Tista URI, which notably includes start and stop times for extracting a part of an archived stream.

5.3 Results

Tista performed well both in a production environment at NRK, and in automated tests. Some problems were reported by users, but none was caused by Tista itself.

5.3.1 Universal addressability

One very interesting aspect of Tista is that it can make any audio stream (even while it is being archived) randomly seekable.

What makes it significantly different from other streaming servers is that the start and stop points are specified as part of a URI. This makes hyperlinking (e.g. to a specific radio programme, song, interview or quote) very easy.

See figure 5.9 for a typical example.

5.3.2 Scalability

Tista scaled well up to 700 simulated concurrent listeners (each listening to a 128 kb/s stream): New users who started a stream did not experience problems or delays neither when connecting nor when streaming.

Beyond 700 listeners the bottleneck seemed to be a lack of bandwidth on the PCI bus (which needed to handle communication with both the network card and the SCSI bus).

After the configured maximum number of users was reached clients could still download data, but at lower rate than the actual streaming rate (i.e. real-time streaming was no longer possible). By reducing the bitrate of the streams the server should therefore be able to scale much further.

At 700 listeners CPU load on the server was around 20%, mostly split between Tista, Apache and MySQL. Detailed CPU load records were unfortunately not kept.

5.3.3 Automatic performance testing

To test performance an automatic testing script (see Appendix B) was run continously for up to 24 hours at full load (i.e. 700 listeners), and all parts of the system (and the system as a whole) remained stable and responsive. For greater realism the script was run on a number of computers at the same time, and not on the one that housed Tista.

The testing script can simulate a realistic production situation by spawning an adjustable number of virtual clients. Each client will sleep for a random number of seconds, connect to Apache, request a list of the 128 most recently broadcast programmes, and randomly select one for download. The

programme is then downloaded in its entirety at maximum speed, before restarting the procedure by sleeping and connecting to Apache again. (Downloaded data are continuously discarded.)

The script can be configured to make its clients either pick a programme's bitrate at random (among the available ones), or always choose a specific rate. Our experience indicates that the majority of real users stream at the highest bitrate; always using the maximum rate therefore probably gives more representative results.

By behaving quite similarily to real clients the script stresses the entire system realistically, not only the Tista streaming server alone. This is useful since it can uncover scalability problems in e.g. the database structure or in the web scripts.

In a real-life situation clients' choice of programmes will probably not be as uniformly distributed as in the test, and will probably be clustered around the most recent or most popular ones. This will most likely improve performance and scalability, since the chances are greater that a requested piece of data is already present in the file system's buffer or the RAID system's cache.

5.3.4 Problems

Tista worked very reliably in a production environment at NRK for eight months (from October 2003 to June 2004), when it was phased out in favour of a Windows Media-based solution. The problems reported by NRK's users fall mostly in the following categories, none of which was caused by Tista itself.

- Problems playing back archived programmes in Windows Media Player (WMP). This seemed to be caused by a bug in popular versions of WMP, and applied to MP3 streaming in general (i.e. the problem was not specific to Tista). The WMP problem eventually caused the change to the WMA-based service.
- Problems caused by missing or incorrect metadata, due to errors or inconsistencies in the data fed to Tista (e.g. programmes with identical start and end times).
- Missing programmes, caused by interruptions of streams that Tista was recording. Because of the way streams are stored, a few minutes worth of missing data would be enough to make the streaming server consider several programmes to be missing (i.e. all programmes completely or partially broadcast during a damaged one-hour chunk). Improvements to the storage routines would probably alleviate this problem.

5.4 Further development

5.4.1 Real-time streaming

Tista has already been extended by Andreas Bergstrøm to support real-time audio streaming. Since this paper describes the older version of Tista *without* such support, the software pointed to in Appendix **??** does not include the real-time streaming code. It can, however, be found at `http://tista.sourceforge.net/`.

5.4.2 Robustness

As previously mentioned Tista has performed very well, but its current data storage model has caused some programmes to be lost.

The most effective improvement to Tista would probably be more error-tolerant storage of streams. For instance, each audio file could be accompanied by an index file containing for every second worth of data a timestamp (e.g. seconds since the platform's epoch) and the position of the corresponding data in the file; it would then be easy to find the right file and position given a date and time, and if any data were lost it would not affect correctly stored data later in the file (assuming that new audio and index data are continuously appended to their respective files). M.Sc. student Jens Remi Karlsen is currently working on implementing such a scheme. (Embedding the index information inside audio files is another option. Unfortunately, this would yield non-standard files that could not easily be used by other applications.)

Tista would naturally also benefit from redundant storage (e.g. running two systems in parallel on separate computers with automatic synchronisation of data lost by one of them) as well as redundant encoding and streaming servers to maximise the chances that valid audio data are available for downloading.

5.4.3 Other formats

The current implementation of Tista can only store and retrieve streams with well-known constant bitrates, since the streaming server calculates file positions based only on a stream's bitrate and a time offset. Handling of variable-rate formats or streams whose bitrates are not precisely known, requires a more sophisticated approach, like the method outlined in the previous section.

Media players also expect streams in many formats to begin with specific headers and/or at specific points in the stream (e.g. at frame boundaries). Tista currently only supports seeking to frame boundaries in MP3 streams.

It should be relatively easy to extend Tista to support formats like Ogg Vorbis and AAC, and even video formats like MPEG-4.

5.4.4 Other protocols

Tista uses HTTP both for receiving streams (for archiving) and for streaming audio to clients. While HTTP is not an ideal streaming protocol it has important strengths in its simplicity and familiarity.

An obvious improvement of Tista would be using RTSP [53] for sending data. At least two kinds of solutions could be imagined:

1. Including an RTSP server in the Tista streaming server itself.

2. Integrating Tista with an external RTSP server, such as Darwin Streaming Server [78] (DSS).

The former option would be more elegant but requires a robust RTSP module (either pure Python code or a linkable library) that can easily be integrated with the rest of the streaming server. I have not come across any such modules.

The latter option might seem a bit of a hack, but would make leveraging a well-known and well-tested RTSP (e.g. DSS) quite simple. For instance, when receiving a request for a programme Tista could extract the corresponding data (using the same functions as when streaming using HTTP), store the assembled programme as a file on disk and refer the client to the file via the RTSP server. A significant disadvantage with this method is that it would be difficult to offer a programme for streaming before airing has ended.

5.4.5 User interface

Tista currently has a Spartan user interface, designed to be used by administrators only. (The interface seen by regular users was implemented separately by NRK).

System settings are specified in an XML file (config.xml, see Appendix D), with the exception of a few options set directly in the Python code itself. Starting, stopping and debugging are performed via the UNIX shell. Some runtime information (e.g. listings of connected users) are avilable via Tista's web interface.

There is thus a lot of potential in improving and expanding the user interfaces for both administrators and users.

5.4.6 Scalability

As previously discussed, Tista scales well up to several hundred simultaneous listeners on a relatively inexpensive computer. It should be trivial to increase this number a bit further by clustering of servers or by using more advanced hardware. Also, reducing the average bitrate of the streams being served (by reducing quality or using more efficient compression, or both) can significantly improve the capacity.

It seems likely that the popularity of on-demand delivery of media will increase significantly in the near future. Creating a streaming system capable of scaling to a really large number of concurrent users (i.e. hundreds of thousand or millions) is certainly an interesting challenge. Tista should lend itself well to experimentation with e.g. schemes for peer-to-peer distribution. (BBC's plans for its Creative Archive project[79] are a strong indication that solutions for scalable distribution of high-quality digital media content will be increasingly important in the future.)

Storing MP3-compressed audio is not very demanding. For instance one channel of 128 kbps data consumes about 1.3 GB of disk space per day, or 470 GB per year. Storing a year's worth of data is thus almost possible using only one desktop IDE drive[11]. Even when considering the added complexity and cost of a RAID system (for greater robustness and streaming capacity) storage is not an important limiting factor. Indeed, if we assume that hard disk capacity per unit of cost continues to double roughly every twelve months for a few more years permanent and complete archiving of radio channels should be within easy reach.

[11] On 14 May 2005 the highest-capacity hard disk available at the Komplett.no web store was a 400 GB serial ATA drive costing NOK 2450.

Permanent archiving of uncompressed audio or TV-quality (or better) video is still not trivial. Also, any system designed to serve a very large archive to a very large number of users would need efficient routines for replication, caching and distribution of data.

5.5 Resources

Planning of Tista as part of NRK's new radio on demand service began in March 2003. Most of the system was implemented during April and the May. Systematic testing began in June. couple of weeks.

Only minor changes were made during the late summer and autumn, mostly to address integration issues as NRK's project team worked on the front end.

Unfortunately, detailed and reliable records of the time used for different parts of the project are not available.

Chapter 6

Conclusion

This chapter suggests future directions for Pycast and Tista, and concludes with reflecting on the future of streaming and multimedia on the Internet.

6.1 Further plans for Pycast

As previously mentioned the current version of Pycast has a number of shortcomings, and it is quite clear that the application needs to be restructured and cleaned up.

The main priority is to rid Pycast of the unpredictability that currently plagues it. Specifically, an error in a node of a data distribution tree should not be allowed to create problems upstream (see section 4.2.6).

Another important short-term improvement is better support for RTP multicasting. The current RTP module is prone to freezing, and also creates streams that are incompatible with the QuickTime and Real media player (and probably also others).

When time permits we plan to publish Pycast on SourceForge as well. In the longer term we hope to implement the changes described in section 4.2.7.

6.2 Further plans for Tista

We are currently preparing to publish Tista as a SourceForge [80] project and under the GNU General Public License. The version in question is almost identical to the one described in this paper, but with some changes made to the configuration file format (using Python code instead of XML) and without some of the auxilliary scripts specific to its use at the NRK. Also, the notation for extracting audio clips has been modified slightly. Information will be posted at `http://tista.sourceforge.net/`.

It is likely that Tista soon will be used as part of our *Stortinget når det passer* [81] project, which offers video recordings from sessions of the Norwegian Parliament to the public on demand. Tista in its current form would be useful for publishing an audio version for portable MP3 players, including the option of skipping to individual speakers. We later hope to extend Tista with similar functionality for MPEG-2 (and possibly also MPEG-4) video.

Also, a group of B.Sc. students (Erik Bergh, Karl Vegard Hansen and Christian Olszewski) are working on integrating Tista with MediaWiki [82] to create a system for collaborative annotation of media streams.

Finally, it seems like Tista might be very well suited for the emerging field of podcasting [83]: Podcast creators could then easily offer different sections of their shows, and traditional radio stations could use it to easily extract programmes from their continuous streams.

6.3 The future of multimedia on the Internet

These are interesting times for multimedia material on the Web and the Internet in general. On the one hand, the established producers, distributors and owners of various media seek to protect and expand their business models and influence.

On the other new technologies and techniques for creating, distributing and using documents, news, blogs, reference works, movies, music, animation and a myriad of other media, are spreading rapidly among the citizens of the Net. Indeed, this process seems to be accelerating.

The interests of the new and the old frequently collide, especially when copyright and other privileges are involved. As a result important precedents are currently being set, and will strongly influence our future both online and offline.

6.3.1 Observations

Before proceeding to reflecting a bit on the future, I will make a few observations about the state of the Internet, streaming, and relevant technological and social developments.

- There is a growing public understanding that open standards are necessary (or at least very useful) for electronic exchange of many kinds of information, both for compatibility and equality of access. For instance The Norwegian Board of Technology specifically recommends using open standards for streaming of public TV and radio channels [84]. (The report in question was accompanied by a summary in English[85].)

- During the last couple of years the number of people who publish information online seems to have grown dramatically, thanks especially to the phenomenon of blogging. Blogging lowers the bar for participation, and programmable interfaces and APIs like RSS [86], most of which are open, enable powerful network effects.

- Load-distributing peer-to-peer transport protocols like BitTorrent [33] make it possible to publish large files (typically music and movies) in a way that scales well up to hundreds or thousands of recipients. (As always, there is a risk that such technologies can be abused for unauthorised copying; apprehension about the character of content does not detract from the efficiency or success of that technology.)

- An increasing number of diverse computer devices are expected to communicate with each other with a minimum of configuration, user involvement and fuss. This requires standardisation (and support for transcoding material between formats). Some examples: PCs, laptops, mobile phones, MP3 players, still cameras, video cameras, game consoles (portable and stationary), DVD players, CD players, TVs, personal video recorders (PVRs), set-top boxes, PDAs, toys, appliances (refrigerators, microwave ovens, robot vacuum cleaners etc.), cars.

- Technologies such as PVRs, podcasting and on-demand downloading mean that TV and radio programmes can be enjoyed whenever and wherever it is convenient.

- Many publishers and content owners understandably fear unauthorised copying and are adopting various digital rights management (DRM) schemes in an attempt to restrict the ways customers can use the material. DRM is sometimes also used to impose other limitations, for instance blocking the playback of DVD disks that have been bought on another continent, or hindering blind users in feeding the contents of electronic books to speech synthesisers.

6.3.2 Downloading vs. streaming

I would venture to suggest this:

In the near future most TV programmes (and similar content) will be downloaded and watched (or listened to) at the users' leisure, not streamed or broadcast in real time[1]*. One notable exception is breaking news, where real-time feeds are valuable.*

This development is likely for several reasons:

- Downloading increases predictability for receivers, since the process is far less sensitive to bandwidth fluctuations and other network problems.

- Downloaded material can more easily be moved or copied to other devices, such as MP3 players, mobile phones and portable video players.

- Files can be downloaded in the background and played back later, for instance making it possible to distribute high-quality movies over slow network connections.

- The Internet's best-effort nature [87] works very well for copying files, but not as well for streaming. (It can pretty much guarantee that a packet of data will be delivered, but not that it will happen quickly enough to fit in its proper place in a stream while it is being played back.)

Unless one single company can achieve domination in the diverse set of markets involved, hassle-free compatibility and interoperability between different devices and software applications will require open standards.

[1] *Downloading* in this context also includes progressive downloading, where data is transported in a non-lossy way (e.g. using HTTP) and may be stored locally while being played back.

6.4 Final remarks

Tista and Pycast could work well as the core modules of a comprehensive, robust and open broadcast streaming system comprising all manner of multimedia data. Given a comprehensive and well designed outline of such a system, its future development might very well benefit from contributions as part of an open source development process.

Appendix A

Abbreviations

AAC Advanced Audio Coding (audio codec)

ARPA Advanced Research Projects Agency (former name of US military research organisation)

ASM Any-Source Multicast (multicast model)

ATA Advanced Technology Attachment (storage device interface)

ATM Asynchronous Transfer Mode (low-level network protocol)

AV Audio/Video

BBC British Broadcasting Corporation

BOS Beginning Of Stream (Ogg data structure)

BSD Berkeley Software Distribution (software license)

CBR Constant Bitrate (audio or video coding)

CC CSRC Count (RTP)

CD Compact Disc (storage medium)

CGI Common Gateway Interface (Web programming)

CPU Central Processing Unit (computer component)

CSRC Contributing Source Identifiers (RTP)

DAB Digital Audio Broadcast (digital radio standard)

DHCP Dynamic Host Configuration Protocol (network protocol)

DRM Digital Rights Management

DSS Darwin Streaming Server (streaming server)

DVB Digital Video Broadcasting (digital TV standard)

DVD Digital Versatile Disc (storage medium)

FD File Descriptor (file object identifier)

FIFO First In, First Out (filesystem construct)

FLAC Free Lossless Audio Codec (audio codec)

GNU GNU's Not Unix (organisation)

GUI Graphical User Interface

HDTV High-Definition Television (digital TV standard)

HTML HyperText Markup Language (markup language)

HTTP HyperText Transfer Protocol (network protocol)

IDE Integrated Drive Electronics (see ATA)

IETF Internet Engineering Task Force (organisation)

IM Instant Messaging (class of computer programmes)

IP Internet Protocol (network protocol)

ISO International Organization for Standardization (organisation)

IT Information Technology

ITU International Telecommunication Union (organisation)

M3U (playlist format)

MP1 MPEG Audio Layer 1 (audio codec)

MP2 MPEG Audio Layer 2 (audio codec)

MP3 MPEG Audio Layer 3 (audio codec)

MP4 MPEG-4 Part 14 (file format)

MPEG Moving Picture Experts Group (working group)

NASA National Aeronautics and Space Administration (organisation)

NAT Network Address Translation (networking technique)

NFS Network File System (network protocol)

NLS online system (computer system)

NOK Norwegian Kroner (currency)

NRK Norwegian Broadcasting Corporation (*Norsk rikskringkasting*, organisation)

NSC Network Secure Communications (ARPA project)

NTP Network Time Protocol (network protocol)

NVP Network Voice Protocol (network protocol)

NVP-II Network Voice Protocol II (network protocol)

OS Operating System

OSI Open Systems Interconnection Reference Model (network model)

P1 (Norwegian radio channel)

P2 (Norwegian radio channel)

P2P Peer-to-Peer (network)

PC Personal Computer

PCI Peripheral Component Interconnect (computer bus)

PCM Pulse-Code Modulation (audio sampling method)

PDA Personal Digital Assistant (handheld computer)

PES Packetised Elementary Stream (MPEG)

PS Programme Stream (MPEG)

PVP Packet Video Protocol (network protocol)

PVR Personal Video Recorder

RAID Redundant Array of Inexpensive Disks

RFC Request for Comments

RSS (standard for Web syndication)

RTCP Real-Time Control Protocol (network protocol)

RTP Real-Time Transport Protocol (network protocol)

RTSP Real-Time Streaming Protocol (network protocol)

SAP Session Announcement Protocol (network protocol)

SCSI Small Computer System Interface (storage and peripheral device interface)

SDP Session Description Protocol (network protocol)

SQL Structured Query Language (database language)

SSM Source-Specific Multicast (multicast model)

SSRC Synchronisation Source (RTP)

TCP Transmission Control Protocol (network protocol)

TFTP Trivial File Transfer Protocol (network protocol)

TS Transport stream (MPEG)

TV Television

UDP User Datagram Protocol (network protocol)

UML Unified Modeling Language (data modeling notation)

URI Uniform Resource Identifier

UTC Coordinated Universal Time

VBR Variable Bitrate (audio or video coding)

VLC VLC Media Player (media player application)

WMA Windows Media Audio (codec)

WMP Windows Media PLayer (media player application)

XML Extensible Markup Language (markup language)

XMT Extensible MPEG-4 Textual Format (file format)

Appendix B

Source code

B.1 Obtaining the source code

The source code of Tista and Pycast will be made available at `http://www.ia.hiof.no/~audunv/thesis/source/`. Please follow the download instructions found there.

Some copies of this thesis will also include a CD-ROM with the same contents as the URL above. Instructions will be avaiable in the `README.txt` file of the root directory.

Please note that the source code will not necessarily work as-is, since it is likely to represent snapshots of production servers.

B.2 Newer versions

We are planning to publish newer versions of Tista and Pycast at `http://tista.sourceforge.net` and `http://pycast.sourceforge.net`, respectively.

Appendix C

Structure of Tista's metadata database

C.1 About the database

Tista uses a very simple database to store programme metadata, based on the table below. Each record stores information about one programme, broadcast in one specific channel:

channel is the ID of the channel in which the programme was broadcast, e.g. `nrk-p1` or `nrk-alltid-klassisk`.

start is the start time of the programme (local time).

end see *start*.

title is the name of the programme.

abstract is a short description of the programme. This is what e.g. appears a box-out in a newspaper's TV guide.

longDescr is rarely used, but was supposed to hold a more thorough description.

avail is an integer representing a bit-field that holds information about the availability of versions of the programme in different bitrates and formats. This approach (as opposed to using a second table and a *join* operation) yields very efficient queries, at the expense of clarity.

The following convention is used to map the availability of a format to a numerical value:

- MP3, 24 kbit/s; if available: 1
- MP3, 56 kbit/s; if available: 2
- MP3, 128 kbit/s; if available: 4

For instance, if the 24-kbit/s and 128-kbit/s versions are available, the field's value would be 5.

An alternative solution may be to use one of MySQL's built-in types for storing lists.

The primary key is the combination of *channel* and *start*.

C.2 The database structure

This is the SQL statement necessary to create the database's only table:

```
-- MySQL dump 8.22
--
-- Host: localhost    Database: rod3
---------------------------------------------------------
-- Server version       3.23.54

--
-- Table structure for table 'programs'
--

CREATE TABLE programs (
  channel varchar(32) NOT NULL default '',
  start datetime NOT NULL default '0000-00-00 00:00:00',
  end datetime NOT NULL default '0000-00-00 00:00:00',
  title varchar(64) NOT NULL default '',
  abstract text,
  longDescr text,
  avail int(10) unsigned NOT NULL default '0',
  PRIMARY KEY  (channel,start)
) TYPE=MyISAM;
```

Appendix D

Sample Tista configuration file

```
<?xml version="1.0" encoding="iso-8859-1"?>
<config>
  <locations>
    <path>/mnt/rod/disk01</path>
  </locations>

  <server>
    <maxclients>600</maxclients>
    <port>8000</port>
    <address>malxrod01.nrk.no</address>
  </server>

  <limits>
    <maxage>21</maxage>
  </limits>

  <database>
    <name>rod3</name>
    <host>160.68.118.48</host>
  </database>

  <logs>
    <debug>/usr/local/tista/log/tista/debug_log</debug>
    <http>/usr/local/tista/log/tista/http_log</http>
    <tista>/usr/local/tista/log/tista/tista_log</tista>
  </logs>

  <channels>
    <channel>
      <id>nrk-p1</id>
      <name>NRK P1</name>
      <stream>
```

```
    <url>http://radio.hiof.no:8000/nrk-p1-128</url>
    <content-type>audio/mpeg</content-type>
    <ext>mp3</ext>
    <bitrate>128</bitrate>
  </stream>
  <stream>
    <url>http://radio.hiof.no:8000/nrk-p1-56</url>
    <content-type>audio/mpeg</content-type>
    <ext>mp3</ext>
    <bitrate>56</bitrate>
  </stream>
  <stream>
    <url>http://radio.hiof.no:8000/nrk-p1-24</url>
    <content-type>audio/mpeg</content-type>
    <ext>mp3</ext>
    <bitrate>24</bitrate>
  </stream>
</channel>

<channel>
  <id>nrk-p2</id>
  <name>NRK P2</name>
  <stream>
    <url>http://radio.hiof.no:8000/nrk-p2-128</url>
    <content-type>audio/mpeg</content-type>
    <ext>mp3</ext>
    <bitrate>128</bitrate>
  </stream>
  <stream>
    <url>http://radio.hiof.no:8000/nrk-p2-56</url>
    <content-type>audio/mpeg</content-type>
    <ext>mp3</ext>
    <bitrate>56</bitrate>
  </stream>
  <stream>
    <url>http://radio.hiof.no:8000/nrk-p2-24</url>
    <content-type>audio/mpeg</content-type>
    <ext>mp3</ext>
    <bitrate>24</bitrate>
  </stream>
</channel>

<channel>
  <id>nrk-petre</id>
  <name>NRK Petre</name>
  <stream>
    <url>http://radio.hiof.no:8000/nrk-petre-128</url>
    <content-type>audio/mpeg</content-type>
    <ext>mp3</ext>
    <bitrate>128</bitrate>
```

```
      </stream>
      <stream>
        <url>http://radio.hiof.no:8000/nrk-petre-56</url>
        <content-type>audio/mpeg</content-type>
        <ext>mp3</ext>
        <bitrate>56</bitrate>
      </stream>
      <stream>
        <url>http://radio.hiof.no:8000/nrk-petre-24</url>
        <content-type>audio/mpeg</content-type>
        <ext>mp3</ext>
        <bitrate>24</bitrate>
      </stream>
    </channel>

  </channels>
</config>
```

Appendix E

Sample Tista search result

Note: The code below has been slightly edited in order to fit on the page.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<programs>
  <program>
    <channel>nrk-p1</channel>
    <start>2005-03-25T02:00:00</start>
    <end>2005-03-25T02:03:10</end>
    <title>Dagsnytt</title>
    <abstract></abstract>
    <longDescription></longDescription>
    <urls>
      <url>
       http://malxrod01.nrk.no/services/public/make_m3u?\
       channel=nrk-p1&amp;start=2005-03-25T02%3A00%3A00\
       &amp;end=2005-03-25T02%3A03%3A10&amp;bitrate=128
      </url>
      <url>
       http://malxrod01.nrk.no/services/public/make_m3u?\
       channel=nrk-p1&amp;start=2005-03-25T02%3A00%3A00\
       &amp;end=2005-03-25T02%3A03%3A10&amp;bitrate=56
      </url>
      <url>
       http://malxrod01.nrk.no/services/public/make_m3u?\
       channel=nrk-p1&amp;start=2005-03-25T02%3A00%3A00\
       &amp;end=2005-03-25T02%3A03%3A10&amp;bitrate=24
      </url>
    </urls>
  </program>
</programs>
```

Appendix F

Sample Tista log file

F.1 About the log format

F.1.1 Structure

Each line reflects one event, and contains two or more fields separated by space characters:

- Field 1: Timestamp (mandatory): Specifies when the event occurred.
- Field 2: Event class (mandatory): Specifies the type of event; see below.
- Field 3–: Class-dependent information: Describes the event.

CONN, *DISC* and *DEBUG* events are followed by a connection ID (CID) which identifies the connection that caused the event. The CID is taken from a counter that begins at zero every time Tista is started or restarted, and increases by one each time a client connects. When combined with the time of the last Tista start or restart, it uniquely identifies a connection.

F.1.2 Event classes

Events belong to one of the following classes:

START Tista has started (or restarted). The CID is reset to zero.

CONN A client has connected. Field three is the client's CID. Field four includes the channel, start time, end time, format and bitrate of the requested programme, as well as name of the requesting host. The subfield with the value *None* is currently not used.

DISC A client has disconnected. Field three is the CID; the rest of the record specifies how long the client was connected, and the number of bytes transmitted.

DEBUG Debug information. Field three is the CID; the rest of the record describes the event. Logging of debug information can be turned on or off in Tista's configuration file.

ERROR An error record describes an unexpected event that affects Tista globally. The rest of the line describes the error.

F.2 Example

This is a random extract from an actual log file in Tista's custom format (including debug information). All host names have been altered to protect users' privacy, yielding names according to the pattern `xxxxxxxx.example.com`, where `xxxxxxxx` is a string that identifies each host uniquely within the sample.

```
...
2003-11-05T10:45:34 ERROR Error connecting to database. No metadata will be used.
...
2003-11-20T10:46:57 START
...
2003-12-01T10:00:52 CONN 8612 nrk-petre,2003-11-19T20:03:00,2003-11-19T22:00:10,mp3,128,None,9ab02b83.example.com,\
  None,NSPlayer/9.0.0.2980 WMFSDK/9.0
2003-12-01T10:00:52 DEBUG 8612 Opening /mnt/rod/disk01/nrk-petre/2003/11/19/nrk-petre-2003-11-19-19-128.mp3
2003-12-01T10:00:52 DEBUG 8602 Finished - error sending to client ((104, ’Connection reset by peer’))
2003-12-01T10:00:52 DISC 8602 Disconnected (00:23:20, 19380984 bytes)
2003-12-01T10:00:56 DEBUG 8612 Finished - error sending to client ((104, ’Connection reset by peer’))
2003-12-01T10:00:56 DISC 8612 Disconnected (00:00:04, 131072 bytes)
2003-12-01T10:00:57 CONN 8613 nrk-petre,2003-11-12T20:03:00,2003-11-12T22:00:10,mp3,128,None,9ab02b83.example.com,\
  None,NSPlayer/9.0.0.2980 WMFSDK/9.0
2003-12-01T10:00:57 DEBUG 8613 Opening /mnt/rod/disk01/nrk-petre/2003/11/12/nrk-petre-2003-11-12-19-128.mp3
2003-12-01T10:02:29 CONN 8614 nrk-petre,2003-11-27T17:00:00,2003-11-27T18:00:10,mp3,56,None,660c4347.example.com,\
  None,NSPlayer/9.0.0.2980 WMFSDK/9.0
2003-12-01T10:02:29 DEBUG 8614 Opening /mnt/rod/disk01/nrk-petre/2003/11/27/nrk-petre-2003-11-27-16-056.mp3
2003-12-01T10:03:04 CONN 8615 nrk-petre,2003-11-26T17:00:00,2003-11-26T18:00:10,mp3,56,None,660c4347.example.com,\
  None,NSPlayer/9.0.0.2980 WMFSDK/9.0
2003-12-01T10:03:04 DEBUG 8615 Opening /mnt/rod/disk01/nrk-petre/2003/11/26/nrk-petre-2003-11-26-16-056.mp3
2003-12-01T10:03:05 DEBUG 8614 Finished - error sending to client ((104, ’Connection reset by peer’))
2003-12-01T10:03:05 DISC 8614 Disconnected (00:00:36, 500856 bytes)
2003-12-01T10:03:05 DEBUG 8614 Finished - error sending to client ((9, ’Bad file descriptor’))
2003-12-01T10:03:25 CONN 8616 nrk-p2,2003-11-15T15:03:00,2003-11-15T15:40:10,mp3,56,None,90ae945f.example.com,\
  None,NSPlayer/9.0.0.2980 WMFSDK/9.0
2003-12-01T10:03:25 DEBUG 8616 Opening /mnt/rod/disk01/nrk-p2/2003/11/15/nrk-p2-2003-11-15-14-056.mp3
2003-12-01T10:03:27 DEBUG 8600 Opening /mnt/rod/disk01/nrk-petre/2003/11/28/nrk-petre-2003-11-28-22-128.mp3
2003-12-01T10:04:17 CONN 8617 nrk-p2,2003-11-29T15:03:00,2003-11-29T15:40:10,mp3,56,None,90ae945f.example.com,\
  None,NSPlayer/9.0.0.2980 WMFSDK/9.0
2003-12-01T10:04:17 DEBUG 8617 Opening /mnt/rod/disk01/nrk-p2/2003/11/29/nrk-p2-2003-11-29-14-056.mp3
2003-12-01T10:04:18 DEBUG 8616 Finished - error sending to client ((104, ’Connection reset by peer’))
2003-12-01T10:04:18 DISC 8616 Disconnected (00:00:52, 729880 bytes)
2003-12-01T10:04:18 DEBUG 8616 Finished - error sending to client ((9, ’Bad file descriptor’))
2003-12-01T10:04:18 DEBUG 8616 Finished - error sending to client ((9, ’Bad file descriptor’))
2003-12-01T10:04:18 DEBUG 8616 Finished - error sending to client ((9, ’Bad file descriptor’))
2003-12-01T10:04:18 DEBUG 8616 Finished - error sending to client ((9, ’Bad file descriptor’))
2003-12-01T10:04:18 DEBUG 8616 Finished - error sending to client ((9, ’Bad file descriptor’))
2003-12-01T10:04:18 DEBUG 8616 Finished - error sending to client ((9, ’Bad file descriptor’))
2003-12-01T10:04:18 DEBUG 8616 Finished - error sending to client ((9, ’Bad file descriptor’))
2003-12-01T10:05:02 DEBUG 8615 Finished - error sending to client ((104, ’Connection reset by peer’))
2003-12-01T10:05:02 DISC 8615 Disconnected (00:01:58, 1647384 bytes)
2003-12-01T10:05:18 DEBUG 8606 Finished - end reached
2003-12-01T10:05:18 DISC 8606 Disconnected (00:18:45, 35680000 bytes)
2003-12-01T10:05:18 DEBUG 8606 Finished - error calling client.Pump: I/O operation on closed file
2003-12-01T10:05:30 CONN 8618 nrk-p2,2003-11-29T15:03:00,2003-11-29T15:40:10,mp3,128,None,e103c534.example.com,\
  None,NSPlayer/9.0.0.2980 WMFSDK/9.0
2003-12-01T10:05:30 DEBUG 8618 Opening /mnt/rod/disk01/nrk-p2/2003/11/29/nrk-p2-2003-11-29-14-128.mp3
2003-12-01T10:05:52 DEBUG 8601 Opening /mnt/rod/disk01/nrk-petre/2003/11/30/nrk-petre-2003-11-30-09-056.mp3
2003-12-01T10:06:07 CONN 8619 nrk-p2,2003-11-28T18:03:00,2003-11-28T18:58:10,mp3,56,None,6bea92fe.example.com,\
  None,NSPlayer/9.0.0.2980 WMFSDK/9.0
2003-12-01T10:06:07 DEBUG 8619 Opening /mnt/rod/disk01/nrk-p2/2003/11/28/nrk-p2-2003-11-28-17-056.mp3
2003-12-01T10:06:07 DEBUG 8599 Finished - error sending to client ((104, ’Connection reset by peer’))
2003-12-01T10:06:07 DISC 8599 Disconnected (00:36:33, 69720912 bytes)
2003-12-01T10:06:49 DEBUG 8619 Finished - error sending to client ((104, ’Connection reset by peer’))
2003-12-01T10:06:49 DISC 8619 Disconnected (00:00:42, 585608 bytes)
2003-12-01T10:06:54 DEBUG 8603 Opening /mnt/rod/disk01/nrk-petre/2003/11/29/nrk-petre-2003-11-29-12-056.mp3
2003-12-01T10:06:55 CONN 8620 nrk-p1,2003-11-28T15:05:00,2003-11-28T15:59:10,mp3,56,None,6bea92fe.example.com,\
  None,NSPlayer/9.0.0.2980 WMFSDK/9.0
2003-12-01T10:06:55 DEBUG 8620 Opening /mnt/rod/disk01/nrk-p1/2003/11/28/nrk-p1-2003-11-28-14-056.mp3
2003-12-01T10:06:59 DEBUG 8603 Finished - end reached
2003-12-01T10:06:59 DISC 8603 Disconnected (00:28:47, 24012688 bytes)
2003-12-01T10:08:44 CONN 8621 nrk-p2,2003-11-30T09:30:00,2003-11-30T09:56:09,mp3,128,None,a51da6da.example.com,\
  None,NSPlayer/9.0.0.2980 WMFSDK/9.0
2003-12-01T10:08:44 DEBUG 8621 Opening /mnt/rod/disk01/nrk-p2/2003/11/30/nrk-p2-2003-11-30-08-128.mp3
2003-12-01T10:11:14 DEBUG 8613 Finished - error sending to client ((104, ’Connection reset by peer’))
2003-12-01T10:11:14 DISC 8613 Disconnected (00:10:17, 19658248 bytes)
2003-12-01T10:11:14 DEBUG 8613 Finished - error sending to client ((9, ’Bad file descriptor’))
2003-12-01T10:12:13 DEBUG 8605 Opening /mnt/rod/disk01/nrk-petre/2003/11/30/nrk-petre-2003-11-30-08-128.mp3
2003-12-01T10:12:18 DEBUG 8605 Finished - end reached
2003-12-01T10:12:18 DISC 8605 Disconnected (00:28:39, 54886144 bytes)
2003-12-01T10:12:18 DEBUG 8605 Finished - error calling client.Pump: I/O operation on closed file
2003-12-01T10:12:18 DEBUG 8605 Finished - error calling client.Pump: I/O operation on closed file
2003-12-01T10:14:28 CONN 8622 nrk-p2,2003-11-30T16:03:00,2003-11-30T16:30:10,mp3,56,None,88a2984b.example.com,\
  None,NSPlayer/9.0.0.2980 WMFSDK/9.0
2003-12-01T10:14:28 DEBUG 8622 Opening /mnt/rod/disk01/nrk-p2/2003/11/30/nrk-p2-2003-11-30-15-056.mp3
2003-12-01T10:14:33 DEBUG 8622 Finished - error sending to client ((104, ’Connection reset by peer’))
```

```
2003-12-01T10:14:33 DISC 8622 Disconnected (00:00:04, 68475 bytes)
2003-12-01T10:14:33 DEBUG 8622 Finished - error sending to client ((9, 'Bad file descriptor'))
2003-12-01T10:16:54 CONN 8623 nrk-p1,2003-11-29T02:03:00,2003-11-29T03:57:10,mp3,128,None,3a4ace86.example.com,\
  None,NSPlayer/9.0.0.2980 WMFSDK/9.0
2003-12-01T10:16:54 DEBUG 8623 Opening /mnt/rod/disk01/nrk-p1/2003/11/29/nrk-p1-2003-11-29-01-128.mp3
2003-12-01T10:17:49 DEBUG 8607 Opening /mnt/rod/disk01/nrk-petre/2003/11/29/nrk-petre-2003-11-29-12-056.mp3
2003-12-01T10:17:54 DEBUG 8607 Finished - end reached
2003-12-01T10:17:54 DISC 8607 Disconnected (00:28:50, 24012688 bytes)
2003-12-01T10:18:39 DEBUG 8623 Finished - error sending to client ((104, 'Connection reset by peer'))
2003-12-01T10:18:39 DISC 8623 Disconnected (00:01:45, 3356180 bytes)
2003-12-01T10:19:12 CONN 8624 nrk-p1,2003-11-27T22:05:00,2003-11-27T23:59:10,mp3,128,None,3a4ace86.example.com,\
  None,NSPlayer/9.0.0.2980 WMFSDK/9.0
2003-12-01T10:19:12 DEBUG 8624 Opening /mnt/rod/disk01/nrk-p1/2003/11/27/nrk-p1-2003-11-27-21-128.mp3
2003-12-01T10:19:43 DEBUG 8624 Finished - error sending to client ((104, 'Connection reset by peer'))
2003-12-01T10:19:43 DISC 8624 Disconnected (00:00:31, 974204 bytes)
2003-12-01T10:19:43 DEBUG 8624 Finished - error sending to client ((9, 'Bad file descriptor'))
2003-12-01T10:19:43 CONN 8625 nrk-p1,2003-11-22T02:03:00,2003-11-22T03:57:10,mp3,128,None,3a4ace86.example.com,\
  None,NSPlayer/9.0.0.2980 WMFSDK/9.0
2003-12-01T10:19:43 DEBUG 8625 Opening /mnt/rod/disk01/nrk-p1/2003/11/22/nrk-p1-2003-11-22-01-128.mp3
2003-12-01T10:21:03 CONN 8626 nrk-petre,2003-11-30T12:03:00,2003-11-30T13:00:10,mp3,128,None,845fe1a1.example.com,\
  None,NSPlayer/9.0.0.2980 WMFSDK/9.0
2003-12-01T10:21:03 DEBUG 8626 Opening /mnt/rod/disk01/nrk-petre/2003/11/30/nrk-petre-2003-11-30-11-128.mp3
2003-12-01T10:21:03 CONN 8627 nrk-petre,2003-11-30T12:03:00,2003-11-30T13:00:10,mp3,128,None,845fe1a1.example.com,\
  None,NSPlayer/9.0.0.2980 WMFSDK/9.0
2003-12-01T10:21:03 DEBUG 8627 Opening /mnt/rod/disk01/nrk-petre/2003/11/30/nrk-petre-2003-11-30-11-128.mp3
2003-12-01T10:21:03 DEBUG 8626 Finished - error sending to client ((104, 'Connection reset by peer'))
2003-12-01T10:21:03 DISC 8626 Disconnected (00:00:00, 11680 bytes)
2003-12-01T10:21:03 DEBUG 8626 Finished - error sending to client ((9, 'Bad file descriptor'))
2003-12-01T10:21:56 DEBUG 8621 Finished - end reached
2003-12-01T10:21:56 DISC 8621 Disconnected (00:13:12, 25104000 bytes)
2003-12-01T10:22:26 CONN 8628 nrk-p2,2003-11-30T13:03:00,2003-11-30T13:30:10,mp3,56,None,9fc50fa9.example.com,\
  None,NSPlayer/9.0.0.2980 WMFSDK/9.0
2003-12-01T10:22:26 DEBUG 8628 Opening /mnt/rod/disk01/nrk-p2/2003/11/30/nrk-p2-2003-11-30-12-056.mp3
2003-12-01T10:22:50 CONN 8629 nrk-p2,2003-11-30T08:30:00,2003-11-30T09:00:10,mp3,128,None,9e3782f7.example.com,\
  None,NSPlayer/9.0.0.2980 WMFSDK/9.0
2003-12-01T10:22:50 DEBUG 8629 Opening /mnt/rod/disk01/nrk-p2/2003/11/30/nrk-p2-2003-11-30-07-128.mp3
2003-12-01T10:22:59 DEBUG 8617 Finished - end reached
2003-12-01T10:22:59 DISC 8617 Disconnected (00:18:41, 15610000 bytes)
2003-12-01T10:23:48 CONN 8630 nrk-p1,2003-11-29T17:03:00,2003-11-29T18:00:10,mp3,56,None,3842567a.example.com,\
  None,NSPlayer/9.0.0.2980 WMFSDK/9.0
2003-12-01T10:23:48 DEBUG 8630 Opening /mnt/rod/disk01/nrk-p1/2003/11/29/nrk-p1-2003-11-29-16-056.mp3
2003-12-01T10:24:11 DEBUG 8618 Finished - end reached
2003-12-01T10:24:11 DISC 8618 Disconnected (00:18:41, 35680000 bytes)
2003-12-01T10:30:44 DEBUG 8629 Finished - error sending to client ((104, 'Connection reset by peer'))
2003-12-01T10:30:44 DISC 8629 Disconnected (00:07:54, 15059008 bytes)
2003-12-01T10:31:19 DEBUG 8620 Finished - error sending to client ((104, 'Connection reset by peer'))
2003-12-01T10:31:19 DISC 8620 Disconnected (00:24:24, 20357040 bytes)
2003-12-01T10:31:23 CONN 8631 nrk-p1,2003-11-28T15:05:00,2003-11-28T15:59:10,mp3,56,None,6bea92fe.example.com,\
  None,NSPlayer/9.0.0.2980 WMFSDK/9.0
2003-12-01T10:31:23 DEBUG 8631 Opening /mnt/rod/disk01/nrk-p1/2003/11/28/nrk-p1-2003-11-28-14-056.mp3
2003-12-01T10:33:10 CONN 8632 nrk-petre,2003-11-28T19:00:00,2003-11-28T21:00:10,mp3,56,None,5f959aa3.example.com,\
  None,NSPlayer/9.0.0.2980 WMFSDK/9.0
2003-12-01T10:33:10 DEBUG 8632 Opening /mnt/rod/disk01/nrk-petre/2003/11/28/nrk-petre-2003-11-28-18-056.mp3
2003-12-01T10:33:20 CONN 8633 nrk-p1,2003-11-27T02:03:00,2003-11-27T03:57:10,mp3,56,None,002307ea.example.com,\
  None,Nullsoft Winamp3 version 3.0d build 488
2003-12-01T10:33:20 DEBUG 8633 Opening /mnt/rod/disk01/nrk-p1/2003/11/27/nrk-p1-2003-11-27-01-056.mp3
2003-12-01T10:33:22 CONN 8634 nrk-p1,2003-11-27T02:03:00,2003-11-27T03:57:10,mp3,56,None,002307ea.example.com,\
  None,Nullsoft Winamp3 version 3.0d build 488
2003-12-01T10:33:22 DEBUG 8634 Opening /mnt/rod/disk01/nrk-p1/2003/11/27/nrk-p1-2003-11-27-01-056.mp3
2003-12-01T10:33:23 DEBUG 8633 Finished - error sending to client ((32, 'Broken pipe'))
2003-12-01T10:33:23 DISC 8633 Disconnected (00:00:02, 37960 bytes)
2003-12-01T10:33:23 DEBUG 8633 Finished - error sending to client ((9, 'Bad file descriptor'))
2003-12-01T10:33:43 DEBUG 8600 Opening /mnt/rod/disk01/nrk-petre/2003/11/28/nrk-petre-2003-11-28-23-128.mp3
2003-12-01T10:33:55 DEBUG 8632 Finished - error sending to client ((104, 'Connection reset by peer'))
2003-12-01T10:33:55 DISC 8632 Disconnected (00:00:45, 629408 bytes)
2003-12-01T10:34:09 CONN 8635 nrk-petre,2003-11-30T09:03:00,2003-11-30T11:00:10,mp3,56,None,5f959aa3.example.com,\
  None,NSPlayer/9.0.0.2980 WMFSDK/9.0
2003-12-01T10:34:09 DEBUG 8635 Opening /mnt/rod/disk01/nrk-petre/2003/11/30/nrk-petre-2003-11-30-08-056.mp3
2003-12-01T10:34:44 CONN 8636 nrk-petre,2003-11-30T09:03:00,2003-11-30T11:00:10,mp3,128,None,6f218764.example.com,\
  None,NSPlayer/9.0.0.2980 WMFSDK/9.0
2003-12-01T10:34:44 DEBUG 8636 Opening /mnt/rod/disk01/nrk-petre/2003/11/30/nrk-petre-2003-11-30-08-128.mp3
2003-12-01T10:34:48 DEBUG 8635 Finished - error sending to client ((104, 'Connection reset by peer'))
2003-12-01T10:34:48 DISC 8635 Disconnected (00:00:39, 541808 bytes)
...
```

Appendix G

Images of HiØ's current streaming system

This appendix contains screenshots of web pages related to HiØ's current radio streaming system, as well as pictures of its various hardware components.

Figure G.1: Front page (`http://media.hiof.no/`)

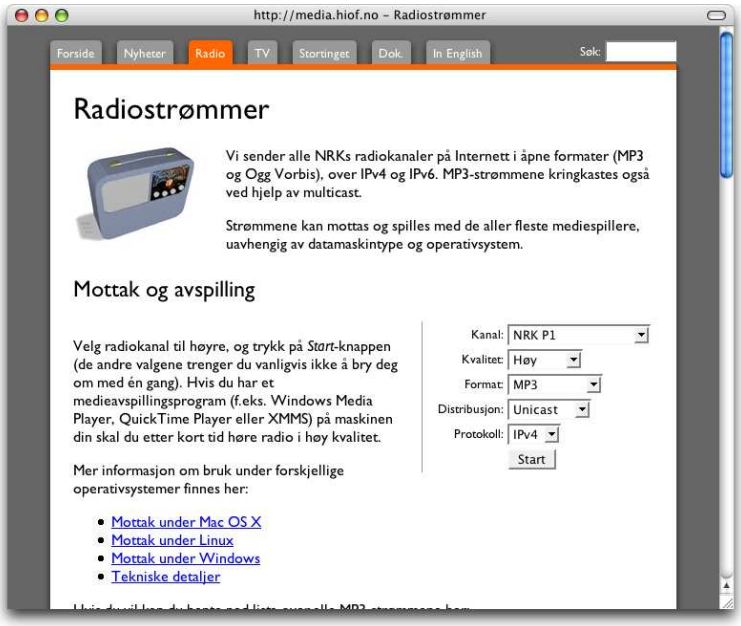

Figure G.2: Radio streaming (`http://media.hiof.no/radio/`)

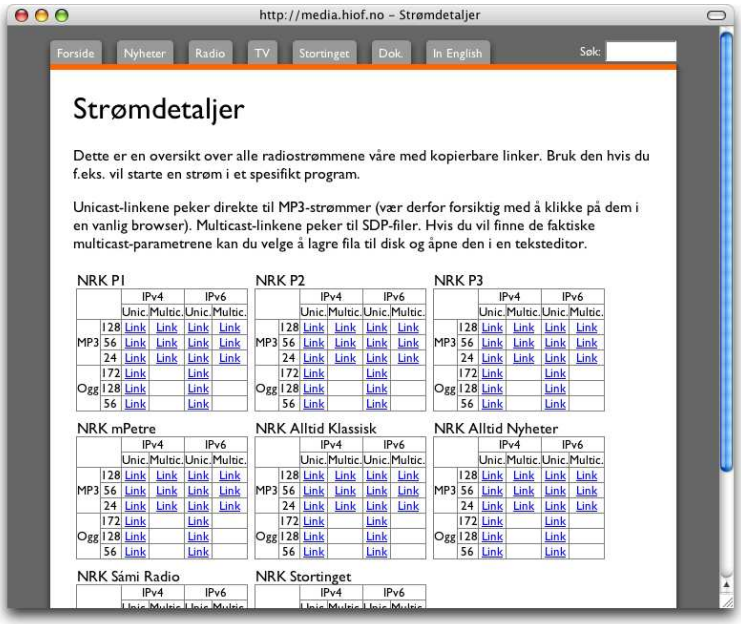

Figure G.3: Technical details (`http://media.hiof.no/scripts/stream_details.php`)

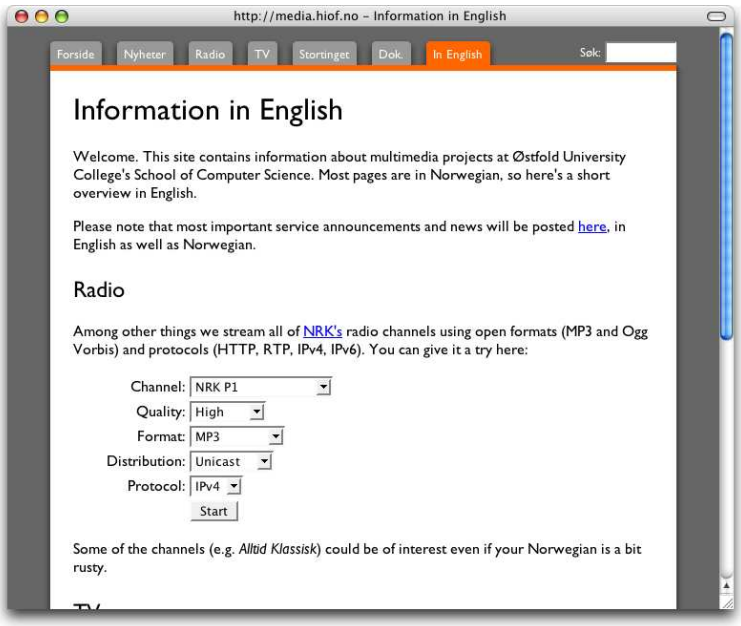

Figure G.4: Information in English (`http://media.hiof.no/english/`)

Figure G.5: News and service announcements (`http://media.hiof.no/news/`)

Figure G.6: School of Computer Science building

Figure G.7: Satellite antennas on the roof. The nearest lower one is used for data reception. The nearest one on the top is used for tuning and testing.

Figure G.8: Eight-way antenna head

Figure G.9: DAB antenna, mounted on south wall

Figure G.10: Server room

Figure G.11: Primary Pycast encoder (contains DVB card)

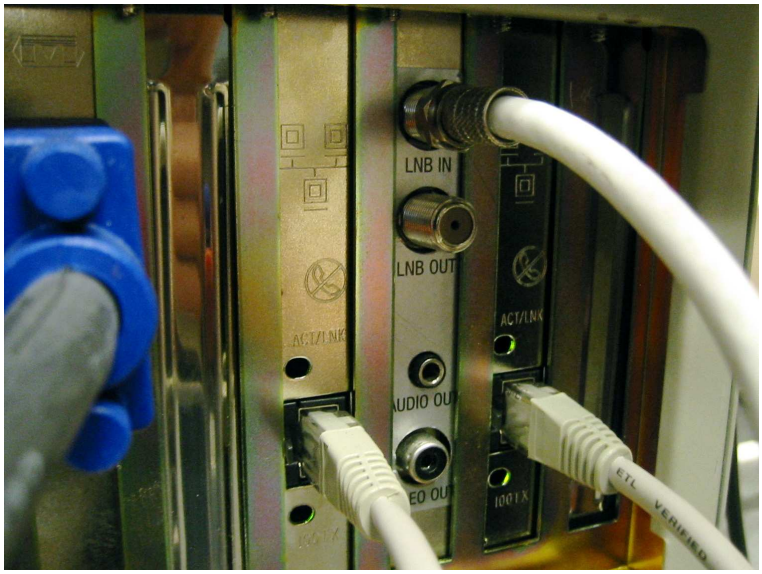

Figure G.12: DVB interface, primary encoder

Figure G.13: Secondary Pycast encoders (diskless cluster nodes)

Figure G.14: NRK P1 encoder

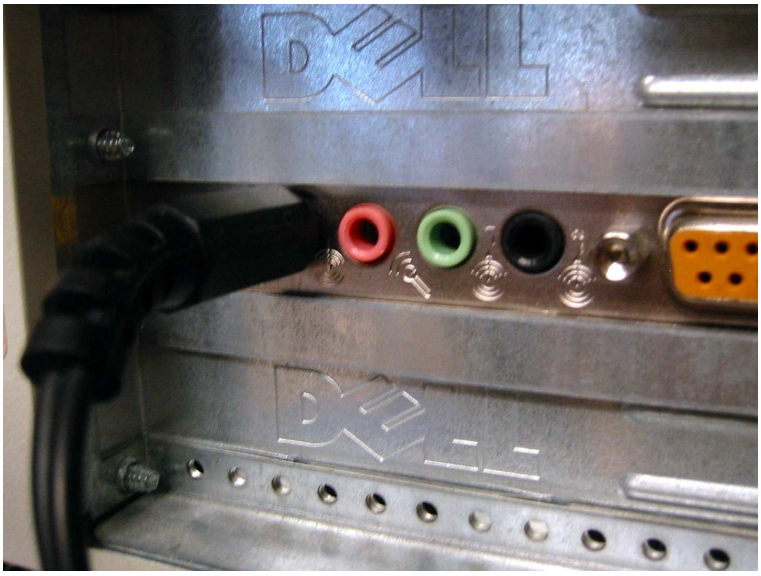

Figure G.15: Sound card, NRK P1 encoder

Figure G.16: DAB tuner (feeds NRK P1 encoder)

Figure G.17: Rear of DAB tuner

Bibliography

[1] Icecast. http://www.icecast.org/.

[2] VLC media player. http://videolan.org/vlc/.

[3] QuickTime. http://www.apple.com/quicktime/.

[4] Windows Media. http://www.microsoft.com/windows/windowsmedia/default.aspx.

[5] D. C. Engelbart and W. K. English. A research center for augmenting human intellect. In *AFIPS Conference Proceedings of the 1968 Fall Joint Computer Conference*, pages 395–410, December 1968. Available online at http://bootstrap.org/augdocs/friedewald030402/researchcenter1968/ResearchCenter1968.html.

[6] Wikipedia, the free encyclopedia. NLS (computer system). http://en.wikipedia.org/wiki/NLS_(computer_system), revised 20:33, 25 March, March 2005.

[7] D. Cohen. Specifications for the Network Voice Protocol (NVP). RFC 741, January 1976.

[8] D. Cohen. *A Network Voice Protocol NVP-II*. USC/Information Sciences Institute, 1981.

[9] E. Cole. *PVP - A Packet Video Protocol*. USC/Information Sciences Institute, 1981.

[10] S. Casner and S. Deering. First IETF Internet audiocast. *ACM SIGCOMM Computer Communications Review*, 22(3), July 1992.

[11] S. Deering. Host extensions for IP multicasting. RFC 1112, August 1989.

[12] K. Almeroth. Evolution of multicast: From the MBone to inter-domain multicast to Internet2 deployment. *IEEE Networks*, January/February 2000.

[13] NASA TV. http://www.nasa.gov/multimedia/nasatv/.

[14] C. Diot K. Almeroth, S. Bhattacharyya. Challenges of integrating ASM and SSM ip multicast protocol architectures. In *International Workshop on Digital Communications: Evolutionary Trends of the Internet (IWDC'01)*, September 2001.

[15] S. Bhattacharyya. An overview of source-specific multicast (SSM). RFC 3569, July 2003.

[16] T. Dorcey. CU-SeeMe desktop videoconferencing software. *Connexions*, 9(3), March 1995. Available online at `http://myhome.hanafos.com/~soonjp/dorcey.html`.

[17] M. Sattler. *Internet TV With CU-SeeMe*. Sams, September 1995. A pre-print version is available online at `http://www.geektimes.com/michael/CU-SeeMe/internetTVwithCUSeeMe/`.

[18] H. Kise Jr. Sanntids multimediakommunikasjon i uninett. *HØit*, 1996.

[19] B. Ludvigsen. A home on the WEB. In *First International Conference on the World-Wide Web*, May 1994.

[20] ivisit. `http://www.ivisit.com/`.

[21] José Alvear. *Guide to Streaming Multimedia*. John Wiley & Sons, April 1998.

[22] Wikipedia, the free encyclopedia. RealNetworks. `http://en.wikipedia.org/wiki/RealNetworks`, revised 20:02, 27 April, April 2005.

[23] Microsoft wins industry support for windows media technologies 4.0. `http://web.archive.org/web/19991205184109/http://www.microsoft.com/presspass/press/1999/Apr99/Streamsupppr.htm`, April 1999.

[24] Apple releases quicktime 4 public beta with internet streaming. `http://web.archive.org/web/19991117114516/http://apple.com/pr/library/1999/apr/19qt4beta.html`, April 1999.

[25] Winamp. `http://winamp.com/`.

[26] Shoutcast. `http://www.shoutcast.com/`.

[27] New web radio technology aims for masses. `http://news.com.com/2100-1023-219634.html`, December 1998.

[28] Rob Brunner. Radio daze. *Entertainment Weekly*, page 102, April 1999.

[29] Ogg Vorbis streams from Virgin Radio. `http://www.virginradio.co.uk/thestation/listen/ogg.html`.

[30] NRK.no — lyd. `http://www.nrk.no/lyd/`.

[31] S. Deering and R. Hinden. Internet protocol, version 6 (IPv6) specification. RFC 2460, December 1998.

[32] Wikipedia, the free encyclopedia. Network address translation. `http://en.wikipedia.org/wiki/Network_address_translation`, revised 23:53, 6 May, May 2005.

[33] BitTorrent. `http://www.bittorrent.com/`.

[34] BitTorrent. http://web.archive.org/web/20010812032735/http://bitconjurer.org/BitTorrent/. Archived version of site, summer 2001.

[35] Norwegians try out tv on mobiles. http://news.bbc.co.uk/2/hi/technology/3829343.stm.

[36] Ices. http://www.icecast.org/ices.php.

[37] J. L. Mitchell, W. B. Pennebaker, C. E. Fogg, and D. J. LeGall. *MPEG Video Compression Standard*. Chapman & Hall, 1996.

[38] K. Brandenburg. MP3 and AAC explained. In *The Proceedings of the AES 17th International Conference: High-Quality Audio Coding*. Fraunhofer Institute for Integrated Circuits FhG-IIS A, 1999.

[39] F. Pereira and T. Ebrahimi, editors. *The MPEG-4 Book*. Prentice Hall PTR, 2002.

[40] J. Moffitt. Ogg Vorbis—open, free audio—set your media free. *Linux Journal*, 2001.

[41] Wikipedia, the free encyclopedia. Vorbis. http://en.wikipedia.org/wiki/Vorbis, revised 8 February, 2005.

[42] Wikipedia, the free encyclopedia. Ogg. http://en.wikipedia.org/wiki/Ogg, revised 8 May, 2005.

[43] Terry Pratchett. *Small Gods (A Discworld Novel)*. 1992.

[44] S. Pfeiffer. The Ogg encapsulation format version 0. RFC 3533, May 2003.

[45] RealNetworks. http://www.realnetworks.com/.

[46] Helix community. https://helixcommunity.org/.

[47] BBC R&D — Dirac. http://www.bbc.co.uk/rd/projects/dirac/.

[48] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol — HTTP/1.1. RFC 2616, June 1999.

[49] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A transport protocol for real-time applications. RFC 3550, July 2003.

[50] M. Handley, C. Perkins, and E. Whelan. Session announcement protocol. RFC 2974, October 2000.

[51] H. Schulzrinne. RTP profile for audio and video conferences with minimal control. RFC 1890, January 1996.

[52] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear. Address allocation for private internets. RFC 1918, February 1996.

[53] H. Schulzrinne, A. Rao, and R. Lanphier. Real time streaming protocol (RTSP). RFC 2326, April 1998.

[54] M. Handley and V. Jacobson. SDP: Session description protocol. RFC 2327, April 1998.

[55] Audioactive. http://www.audioactive.com/.

[56] openMosix. http://openmosix.sourceforge.net/.

[57] LinuxTV project. http://linuxtv.org/.

[58] Sound eXchange. http://sox.sourceforge.net/.

[59] The LAME project. http://lame.sourceforge.net/.

[60] OggEnc — command line encoder for the Ogg Vorbis format. http://directory.fsf.org/audio/ogg/OggEnc.html.

[61] dvbstream. http://www.linuxstb.org/dvbstream/.

[62] R. Droms. Dynamic host configuration protocol. RFC 2131, March 1997.

[63] K. Sollins. The TFTP protocol (revision 2). RFC 1350, July 1992.

[64] T. Mathisen and J. O. Hoddevik. NRK radio on demand: Prosjektrapport. Project report, Østfold University College, School of computer science, June 2000.

[65] Mysql. http://www.mysql.com/.

[66] The Apache HTTP server project. http://httpd.apache.org/.

[67] Apache/Python integration. http://modpython.org/.

[68] Nexsan technologies. http://www.nexsan.com/.

[69] mmsclient. http://www.geocities.com/majormms/.

[70] A. Rigo. Representation-based just-in-time specialization and the Psyco prototype for Python. In *Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 15–26. ACM Press, 2004.

[71] Python Software Foundation. *Python Library Reference*, 2.3 edition, 2003. Available online at http://www.python.org/doc/2.3/lib/.

[72] itunes. http://www.apple.com/itunes/.

[73] The Apache Software Foundation. *Apache HTTP Server Version 2.0 Documentation*. Available online at http://httpd.apache.org/docs-2.0/.

[74] Dublin Core Metadata Initiative. http://dublincore.org/.

[75] International Organization for Standardization. *ISO 8601:1988. Data elements and interchange formats — Information interchange — Representation of dates and times*. International Organization for Standardization, 1988.

[76] G. Klyne and C. Newman. Date and time on the Internet: Timestamps. RFC 3339, July 2002.

[77] D. L. Mills. Network time protocol (version 3). RFC 1305, March 1992.

[78] Darwin streaming server. http://developer.apple.com/darwin/projects/streaming/.

[79] BBC Creative Archive pioneers new approach to public access rights in digital age. http://www.bbc.co.uk/pressoffice/pressreleases/stories/2004/05_may/26/creative_archive.shtml.

[80] Sourceforge.net. http://sourceforge.net/.

[81] Stortinget når det passer. http://stortinget.hiof.no/.

[82] MediaWiki. http://wikipedia.sourceforge.net/.

[83] Wikipedia, the free encyclopedia. Podcasting. http://en.wikipedia.org/wiki/Podcasting, revised 11:55, 25 March, 2005.

[84] T. Tennøe, editor. *Programvarepolitikk for fremtiden*. Norwegian Board of Technology, November 2004. Available online at http://teknologiradet.no/html/679.htm.

[85] Software policy for the future - executive summary. http://teknologiradet.no/html/679.htm, 2004.

[86] Wikipedia, the free encyclopedia. RSS (protocol). http://en.wikipedia.org/wiki/RSS_(protocol), revised 15:12, 29 March, 2005.

[87] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions in Computer Systems*, 2(4), November 1984.

Colophon

projects | brain | Emacs | LaTeX > thesis.ps

This thesis was written in Emacs and typeset with LaTeXand BibTeX, installed using Fink 0.21.2, on a PowerBook running OS X 10.3. Labour-saving scripts were programmed in Python 2.3. Illustrations were mostly made using OmniGraffle. Graphs were plotted with GNUPlot.

Various other tasks were performed under Debian GNU/Linux 3.0.

The following Web resources were invaluable:

- Archive.org, for digital archaeology.
- Wikipedia, for to-the-point articles on almost anything.
- And Google, of course.