Video on the World Wide Web

Accessing Video from WWW Browsers

Sverre H. Huseby

February 2, 1997

Abstract

This report discusses inclusion of various kinds of video in browser programs for the World Wide Web. It contains description of video representation formats, video transfer on the Internet in general, and mechanisms for extending Web browsers to support initially unknown media types.

A plug-in for Netscape Navigator, capable of displaying inline MPEG movies, is implemented, along with a Java applet for displaying live video captured from a camera connected to a remote computer. The plug-in and the applet show that making video available from Web browsers is indeed possible, and not considerably harder than making a stand-alone video handling program.

Preface

This report documents my work on a master degree in computer science at Department of Informatics (Ifi), University of Oslo (UiO) in 1996 and 1997. The work was done at the University's Center for Information Technology Services (USIT).

... and I wish to thank ...

My internal supervisors have been Fritz Albregtsen and Per Grøttum. My external supervisor, Ingvil Hovig, has not only been a clever advisor managing to make this write-o-fobic finish his report, but she has also been a great friend. I owe her a lot. Another great source of inspiration is Hanne S. Finstad, who has agreed to marry me as soon as I finish this work. :–) Marius Midtvik at USIT has come up with pointers to several relevant documents and Web sites. While I'm at it, I would also like to say hello to Glenn Lines, and thank him for never saying "no" to another beer at the campus pub.

At the contrary, I would *never* like to thank that infamous net-lag that every now and then worked hard to drive me mad while I was searching the Web for information.

A postscript version, and a HTML version of this document, are available at

`http://www.ifi.uio.no/~ftp/publications/cand-scient-theses/SHuseby/`

The same location also contains source code to the implemented programs, along with a pointer to a demonstration page for the Java video applet.

This document was written using `GNU emacs`,
and typeset at 11 pt. by LaTeX.
Figures were created with `xfig`,
while `xv` captured the screenshots.
The HTML-version was created with LaTeX2HTML.

Contents

1 Introduction **1**

2 Video Representation and Compression **5**
2.1 Sampling 5
2.2 Image and Video Compression 6
2.2.1 Rate vs. Distortion 6
2.3 Single Image Compression 7
2.3.1 JPEG 8
2.4 Exploiting Temporal Redundancy 11
2.4.1 ITU-T Recommendations H.261 and H.263 12
2.4.2 MPEG 13
2.5 Discussion 14
2.6 Summary 15

3 Transferring Video on the Internet **17**
3.1 Introduction to TCP/IP Networking 17
3.1.1 Link Layer 18
3.1.2 Network Layer 18
3.1.3 Transport Layer 19
3.1.4 Application Layer 19
3.1.5 Bandwidth 20
3.1.6 One-to-many and Many-to-many 21
3.2 Multicasting and the MBone 22
3.2.1 Session Management 23
3.2.2 Applications 23
3.3 Methods for General Data Transfer 25
3.3.1 File Transfer Protocol (FTP) 25
3.3.2 Hypertext Transfer Protocol (HTTP) 26
3.4 Methods Related to Video Transfer 26
3.4.1 Real-Time Protocol (RTP) 26
3.4.2 CU-SeeMe 27
3.5 Summary 28

4 Solutions for Embedding Video in WWW Browsers **29**
4.1 Uniform Resource Locators (URLs) 29
4.2 Browsers and Document Types 30
4.3 Spawning External Applications 30
4.4 Server Push and Client Pull 31
4.5 Animated GIFs 32

4.6 Extending Browser Source Code . . . 32
4.7 Plug-ins . . . 33
4.8 Java Applets . . . 35
4.8.1 What is Java? . . . 35
4.8.2 Java Applets and Security . . . 37
4.8.3 Using Java for Video . . . 38
4.9 Discussion . . . 38
4.10 Summary . . . 38

5 MPEG Plug-in for Netscape Navigator 41
5.1 Netscape Plug-in API . . . 41
5.2 Choosing an MPEG Decoder . . . 43
5.2.1 mpeg_play-2.3-patched . . . 43
5.2.2 mpeg2play-1.1b . . . 44
5.2.3 Benchmarks . . . 44
5.2.4 Results . . . 46
5.3 Tailoring `mpeg_play` . . . 47
5.3.1 The Client — Server Approach . . . 47
5.3.2 The `mpeg_play` Library API . . . 49
5.3.3 Avoiding the Pitfalls of Parallel Processing . . . 50
5.3.4 On X11 and Colors . . . 51
5.4 Discussion . . . 52
5.5 Summary . . . 53

6 Sending Camera Input to a Java Applet 55
6.1 Network Communication . . . 55
6.2 Video Handling . . . 56
6.3 Java Applet Implementation . . . 57
6.4 Discussion . . . 59
6.5 Summary . . . 60

7 Conclusion 61

A Introduction to Data Compression 63
A.1 Basic Information Theory . . . 63
A.2 Compression Algorithms . . . 64
A.2.1 Statistical Coding . . . 64
A.2.2 Dictionary Based Coding . . . 67

B SHHVid Java Applet Source Code 71

C SHHVid Grabber Source Code 77

D SHHVid Proxy Source Code 103

E Recoding MPEG to JPEG and GIF 119

F Internet Links 121

Bibliography 125

Chapter 1

Introduction

Internet originated from a military research project sponsored by Department of Defense's (DoD) Advanced Research Projects Agency (ARPA) in the late 60s and the 70s. The original ARPANET included military, university and research sites, and a main goal of the project was to investigate how to build networks that would withstand partial outages and still function [1] [2].

In the early 80s, a new set of protocols were developed for use on the ARPANET, the TCP/IP protocol suite (as described in section 3.1 on page 17). The TCP/IP protocol suite is not bound to any particular type of hardware, making it possible to connect any computer to the network, as long as an implementation of the protocols is available. Development of protocols and other standards for the Internet is an open effort; people all over the world participate in extending the functionality of the net, communicating using the Internet itself. Standards are published as "Request for Comments" (RFC) -documents, where the somewhat misleading name is kept for historical reasons [3].

During the last years, the Internet has grown rapidly. Since 1988, the number of hosts connected has doubled each year [4]. In 1990, the first commercial provider of dial-up Internet access got online, opening the network to the non-research community [5].

Over the years, information has been transferred across the Internet using a plethora of different protocols, all requiring separate programs implementing the protocol in question. Also, lots of information have been available as files in local filesystems. Access to several computers, possibly running different operating systems, may have been necessary to reach the information.

Making all available information more accessible has probably been in the minds of several people, but in 1989 and 1990 Tom Berners-Lee[1] of CERN[2] proposed the initiation of a project that would revolutionize the way we access the world of information on the Internet [6]. The World Wide Web[3] was born [7].

Hypertext documents play a central part in the World Wide Web concept. Aided by a *browser* program, users may view documents (popularly named "pages") in which

[1] http://www.w3.org/pub/WWW/People/Berners-Lee/
[2] http://www.cern.ch/
[3] World Wide Web: also known as WWW, the Web, or W3.

highlighted parts "links" to other documents anywhere on the Internet. Pages may contain various media types, most commonly text and images, but also sound, video and 3D graphics, limited only by the capabilities of the browsers.

To many people, WWW *is* the Internet. Nowadays, Internet service providers (ISP) typically equip their new users with a Web browser, and possibly a separate E-mail program in addition to the dial-up software. Non-technical users will depend on these programs, never exploring the parts of the Internet that cannot be reached by their aid. Also Internet veterans now seem to find the WWW a valuable source of easily searchable information, starting a Web browser along with other useful programs at login time.

The increased use of Web browsing programs, makes it a goal to include more of the Internet under the World Wide Web, making new services just a "mouseclick" or a "keypress" away for both novice and advanced users.

Live[4] video is an area that has begun emerging on the Web, and it is relatively new on the Internet itself. An important reason for a late introduction, is that the bandwidth of the lines connecting the Internet networks together, has been too low for transferring video at acceptable rates. Today, aided by compression technologies that preserve reasonable quality even at excessive levels of compression, combined with higher available bandwidths, the Internet has become a promising ground for transferring and sharing live images.

There are several applications for live video on the Internet. Much research is taking place to develop software, hardware and standards for *video conferencing*, in which two or more people may participate in a meeting or a class from possibly distant locations on the globe. Members of a conference may be seated in front of workstations, or they may be located in special conference rooms, equipped with cameras and microphones. In addition to video and audio, software exists to let the participants use a shared whiteboard for illustrations.

An application similar to video conferencing, is *video telephony*, which has been available on the regular phone network for some years. Enabling telephony, with or without video, on the Internet, will drastically reduce the costs of long distance calls for end users, as the price will be limited by the connection to the local ISP.

Television and cinema play important roles in entertaining the 20th century human. A drawback with these media, is that deciding *when* a certain movie or program is viewed, is not left to the viewer, but rather to the provider. The idea behind *video on demand* systems, is to hand this control to the viewer. In the future, the Internet may be the transport medium of such services.

The aim of this report is to describe existing and evolving methods for transferring miscellaneous kinds of video on the Internet, and outline ideas on how to incorporate these methods into the World Wide Web, making the video accessible from current or future Web browsers. Although a natural companion to video in the above mentioned

[4]In this document, "live" refers to representing real-world actions at approximately the same time they occur, while "real-time" means playing at correct speed, either live or as playback.

applications is sound, this report does not focus on that topic.

Chapter 2 gives an overview of video representation; what a video stream is, and how compression is done. The chapter includes an overview of the compression in JPEG, a standard for still images, in addition to brief explanations of H.261 and MPEG, two standards for video representation.

In chapter 3, methods for video transfer and synchronization on the Internet, including multicasting and the MBone, is described. A short introduction to Internet networking is given.

The next chapter describes the current possibilities and future extensions to allow inclusion of video in Web browsers.

Chapter 5 documents the implementation of a simple program that allows playing MPEG videos inside a popular browser.

Chapter 6 describes the implementation of a Java applet and accompaniment C-programs for receiving video from a remote, computer-mounted camera.

The final chapter contains a discussion and a conclusion.

There are six appendixes: The first gives an introduction to general data compression, to aid in the understanding of chapter 2. The three next contain the source code of the Java applet and C-programs described in chapter 6. The next appendix describes how to recode an MPEG-file to JPEG and GIF, while the last appendix contains a collection of Internet resources with relevance in our context.

Chapter 2

Video Representation and Compression

A few years ago, images and video were represented using "de facto" file formats, typically developed by a single organization, with specifications released to the public. Nowadays, international standardization organizations cooperate to come up with international standards for the same purposes, building their decisions on years of research. A couple of standards, such as JPEG for still images and MPEG for video, are fully defined, while others are being worked upon.

This chapter will give an introduction to image and video representation, starting with the sampling process. Compression plays an important role in modern schemes for video representation, so the next two sections focus on image compression, including describing the compression method used in JPEG. After discussing single image compression, the following sections are dedicated to video compression, and how one may exploit similarities in nearby frames of a video sequence. This section includes brief descriptions of the standards H.261 and MPEG.

The reader may want a basic knowledge of data compression before reading this chapter. Consult appendix A or any of it's referenced papers for an introduction.

2.1 Sampling

A video stream, or video sequence, is generated by sampling fixed images of a scene at certain time intervals — temporal[1] sampling. If the sampling frequency is high enough, typically between 20 and 30 images per second, a playback at the same speed will make the eye and the brain see continuous motion pictures.

Each digital image, also called *frame*, is generated by spatial[2] sampling. Using a camera or a scanner, the continuous, real-life image is converted to a grid of *pixels*[3], each having a discrete value, or a set of discrete values, giving a measure for the intensity or color of the small square it represents. For grayscale images, the pixel value is typically represented using eight bits, giving possible values between 0 and

[1] **temporal:** from latin, tempus, "time".
[2] **spatial**: from latin "space". Having to do with space.
[3] **pixel**: short for "picture element".

255 inclusively. The value usually represents the amount of light within the pixel; 0 is black and 255 is white, while the values between give various shades of gray.

Pixels of color images normally consist of three values, describing a color in a certain *color model*. Well known color models include RGB, where the three values represent the red, green and blue color components, and YCbCr (the digital version of YUV [8]), where one value is used for intensity, while the two others are used to represent chrominance. RGB representation is used by most (if not all) color monitors, while YCbCr and other schemes, separating luminance and chrominance, are used in several image representation schemes containing irreversible compression, along with television sets. For more on color models, see for instance [9, chapter 13] or [10, chapter 3].

To sum up, a raw video stream is a sequence of bytes in which a single or a triple of bytes represent a pixel. A sequence of pixels represent a single image, and a sequence of images make up the entire movie.

2.2 Image and Video Compression

A raw video stream tends to be quite demanding when it comes to storage requirements, and demand for network capacity when being transferred between computers. Before being stored or transferred, the raw stream is usually transformed to a representation using compression. When compressing an image sequence, one may consider the sequence a series of independent images, and compress each frame using single image compression methods, or one may use specialized video sequence compression schemes, taking advantage of similarities in nearby frames. The latter will generally compress better, but may complicate handling of variations in network transfer speed.

Compression algorithms may be classified into two main groups, reversible and irreversible. If the result of compression followed by decompression gives a bitwise exact copy of the original for every compressed image, the method is reversible. This implies that no quantizing is done, and that the transform is accurately invertible, i.e. it does not introduce round-off errors.

When compressing general data, like an executable program file or an accounting database, it is extremely important that the data can be reconstructed exactly. For images and sound, it is often convenient, or even necessary to allow a certain degradation, as long as it is not too noticeable by an observer.

2.2.1 Rate vs. Distortion

The reason to introduce loss of quality, is to reduce the bitrate. In general, a higher allowable distortion gives lower bitrate. Often it may be interesting to have some kind of measure for the degradation of the decompressed image compared to the original. There are two classes of comparison measures, subjective and objective.

Subjective measures are performed by letting a group of people do a side by side comparison of the decompressed and the original image. The comparison is done using predefined quality classes, such as “excellent”, “fine”, “passable”, “marginal”,

"inferior" and "unusable" [11].

Objective measures are mathematically or algorithmically oriented. One well known measure, is Root Mean Squared Error (RMSE). Given an $N \times M$ original image f, and a compressed and decompressed image $\hat{f}$, RMSE is calculated according to the following formula [11, section 6.1.4]:

$$RMSE = \sqrt{\frac{1}{NM} \sum_{x=0}^{N-1} \sum_{y=0}^{M-1} [f(x,y) - \hat{f}(x,y)]^2}$$

RMSE is 0 for identical images. Higher values denote higher deviation between the images. Note that low RMSE not necessarily indicates high subjective quality.

Closely related to RMSE, is Peak Signal to Noise Ratio (PSNR), measured in dB. For an eight bit image, with intensity values between 0 and 255, the PSNR is given by [12, page 77]

$$PSNR = 20 \log_{10} \frac{255}{RMSE}$$

The above objective measures build on differences between single pixels in the two images. This gives results not always comparable to subjective measures. Subjectively, we appreciate removal of noise pixels, while smoothing of edges makes the image look like it is out of focus. In the above functions, noise pixel removal and edge smoothing is treated equally.

2.3 Single Image Compression

One of the more popular standards for reversible image compression, is Compuserve's Graphics Interchange Format (GIF) described in [13] for the original 1987-version, and in [14] for the extended 1989-version. GIF compression is done using the Lempel-Ziv-Welch (LZW) algorithm, based on LZ78. Using the term "reversible" when describing GIF may, in some cases, be a misnomer, as images will have to be quantized to 256 colors before being coded. If the original image contains more than 256 colors, it will not be fully reproducible after coding with GIF. Due to the dictionary based coding, the compression performance of GIF is best when coding images containing repeated patterns, as is often the case with computer generated images and simple line drawings.

Most methods for irreversible, or "lossy" digital image compression, consist of three main steps: Transform, quantizing and coding, as illustrated in figure 2.1.

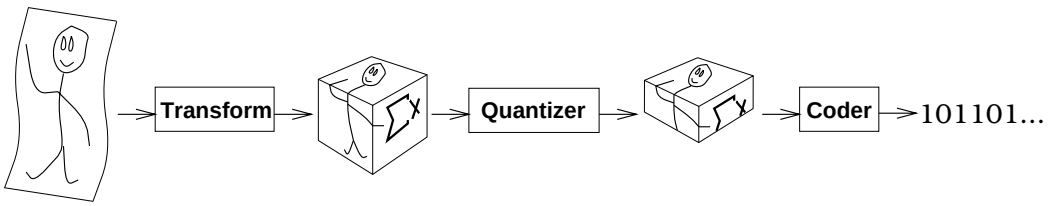

Figure 2.1: The three steps of digital image compression.

The purpose of the transform is to reorganize the data, to make it possible for the encoder to do a better job. For statistical coders, the transform can typically be to give the data a representation featuring non-uniform probability distribution.

The quantizing step is used to remove or reject information that is regarded uninteresting. What is considered uninteresting, depends on how the image is supposed to be used later. If the image is targeted at a human observer, which is the case for the video images covered by this report, the quantizing will typically remove details which are not registered by our visual system.

The final step, coding, produces the resulting bitstream using an appropriate, general compression algorithm.

An irreversible method yields a result after decompression that, using an appropriate quality measure, is close to the original.

2.3.1 JPEG

JPEG is an international standard for color image compression, created from a cooperative effort between the three major standardization organizations ISO[4], CCITT[5] and IEC[6]. The acronym JPEG is short for "Joint Photographic Experts Group". The book [15], written by two members of the standardization working group, is a comprehensive guide to the inner workings of JPEG. It also features a copy of the JPEG draft international standard as an appendix. A shorter introduction to JPEG is given in a "classical" article [16] by Gregory K. Wallace, once chairman of JPEG. A comparison between GIF and JPEG may be found in [17].

JPEG offers many modes of operation with variations in pixel depth, number of color components, color component interleaving, pixel order, and coding algorithm. It even offers a reversible mode. We focus on the way images are treated to make high levels of irreversible compression possible. What is described here, is also relevant for the video sequence compression described in later sections.

The heart of irreversible JPEG, is a 2D version of a mathematical transform known as Discrete Cosine Transform (DCT). The goal of the transform is to decorrelate the original signal, distributing the signal energy to only a small set of coefficients [12]. After the transform, many coefficients may be discarded without, or with little, loss of visual quality.

[4]**ISO**: International Organization for Standardization.
[5]**CCITT**: International Telegraph and Telephone Consultative Committee, now named ITU-T.
[6]**IEC**: International Electrotechnical Commission.

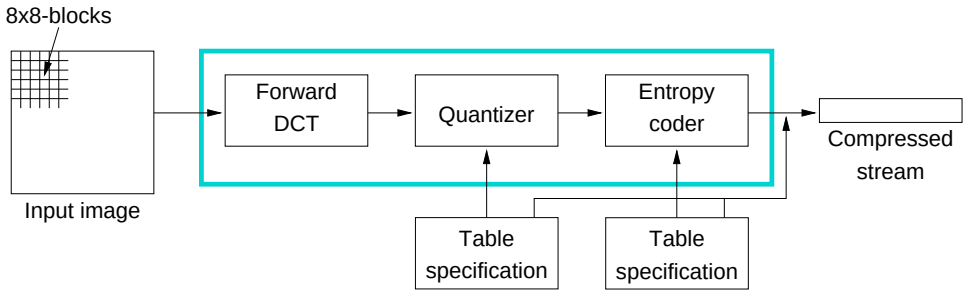

Figure 2.2: Pipeline for DCT-based coding (from the ISO JPEG draft standard [15, Appendix A]).

Figure 2.2 shows the main steps in converting a band (a color component) of an image to a compressed bitstream using DCT-based schemes, such as JPEG. An image is subdivided in *blocks* of 8×8 pixels, each of which are handled independently.

In the JPEG standard, the forward transform (FDCT) and the corresponding inverse transform (IDCT) to be performed on each block (matrix), are defined as

$$\text{FDCT:} \qquad S_{vu} = \frac{1}{4} C_u C_v \sum_{x=0}^{7} \sum_{y=0}^{7} s_{yx} \cos \frac{(2x+1)u\pi}{16} \cos \frac{(2y+1)v\pi}{16}$$

$$\text{IDCT:} \qquad s_{yx} = \frac{1}{4} \sum_{u=0}^{7} \sum_{v=0}^{7} C_u C_v S_{vu} \cos \frac{(2x+1)u\pi}{16} \cos \frac{(2y+1)v\pi}{16}$$

$$\text{where:} \qquad C_z = \begin{cases} \frac{1}{\sqrt{2}} & \text{for } z = 0 \\ 1 & \text{for } z \neq 0 \end{cases}$$

When using FDCT, blocks are transformed, to 8×8 matrixes of transform coefficients. Figure 2.3 illustrates the naming conventions for transform coefficients.

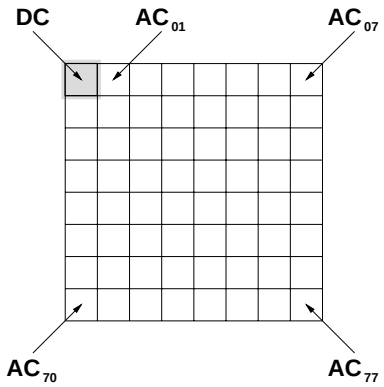

Figure 2.3: DC- and AC-coefficients.

The DC coefficient is proportional to the average pixel value in the original block. AC-coefficients close to DC represent highly correlated pixel values (low frequency),

while AC-coefficients towards the lower right corner represent rapidly changing pixel values (high frequency), such as edges and noise. Using FDCT, most of the energy is collected in the coefficients near DC, with decreasing energy levels towards the lower right AC_{77}-coefficient, i.e. the upper left coefficients are more important to visual quality when restoring the image.

Quantization is done by dividing and truncating each of the transformed coefficients by individual values. The values are given in a *quantization matrix*, which becomes a part of the compressed stream[7] (the leftmost "table specification" in figure 2.2). Quantization is the greatest source to loss of information, as decimal digits are discarded in the truncation. The quantization matrix typically contains higher values towards the lower right, giving several of the less important coefficients a zero value.

Before coding, the quantized block is converted to a sequence of numbers by collecting coefficients according to the zig-zag sequence in figure 2.4.

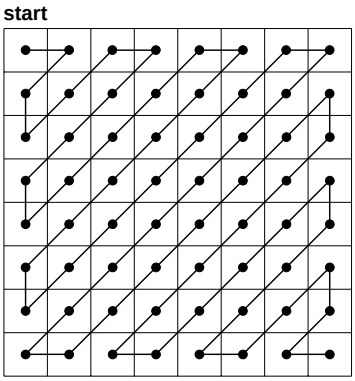

Figure 2.4: The zig-zag sequence.

The zig-zag sequence orders the coefficients in approximately decreasing importance, collecting the more heavily quantized values towards the end. This ordering typically gives runs of zero values, which are runlength encoded. Non-zero values are coded using either Huffman or arithmetic coding. A Huffman code table or an arithmetic coding decision table is sent as part of the compressed stream (the rightmost "table specification" in figure 2.2).

Decompressing a JPEG stream to an image resembling the original, is done using the pipeline in figure 2.5. The process is the reverse of coding.

[7] JPEG allows an "abbreviated format for compressed image data" in which no tables are coded. This may be used between cooperating applications, where tables are predefined.

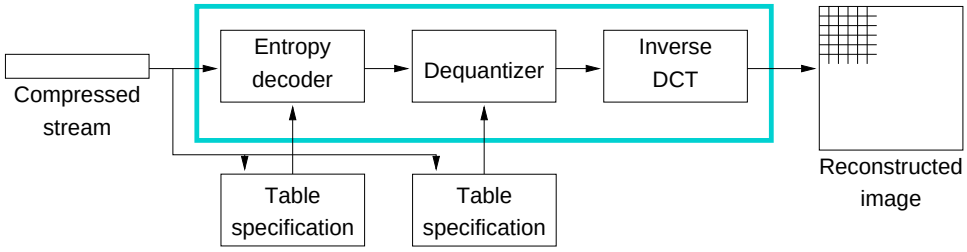

Figure 2.5: Pipeline for DCT-based decoding (from the ISO JPEG draft standard [15, Appendix A]).

Tables for the Huffman or arithmetic decoder is read from the stream, along with quantization matrixes for the dequantizer. After decoding, the dequantizer multiplies the DCT coefficients with the values found in the quantization matrix, before sending the matrix to the inverse DCT. Running IDCT results in an 8×8 block, a part of the reconstructed image.

The JPEG standard doesn't specify how color images are supposed to be split in components. An advisory part of the standard does, however, specify sample quantization matrixes for intensity bands, and chrominance bands. As the human visual system is more sensitive to intensity changes than variations in colors, chrominance bands may be quantized more than intensity bands. JPEG File Interchange Format (JFIF) [18] specifies the use of the YCbCr color model for coded images.

JPEG has been used for video compression, by individually compressing each frame of the video stream. JPEG used for video sequences is often referred to as "motion JPEG" or "M-JPEG", but there is no agreed upon standard for this kind of compression. Different vendors have taken different approaches, with incompatible results [19].

2.4 Exploiting Temporal Redundancy

Considering a movie as a sequence of single, independent images, leaves us without the opportunity to exploit the temporal redundancy: Often there are small changes from frame to frame within a video sequence. The background may be fixed while an object is moving in front of it, or the camera may sweep over a scene, shifting the entire view in one direction.

Standardized compression algorithms exists, taking advantage of similarities between nearby frames. The algorithms typically divide a frame in blocks of 8×8 pixels, and encode each block using discrete cosine transform (DCT). To take advantage of the temporal redundancy, the pixel values in a block may be predicted based on blocks in nearby frames. When such prediction is used, the block is represented not by the actual pixel values, but rather by the differences from the matching pixel values in the frame used for prediction.

To make prediction better, motion compensation is often used: A displacement vector may be associated with a block, describing how the block has moved relatively to the frame used for prediction. The vector should point to the block giving optimal

prediction. The task of finding the optimal block when coding, is computationally expensive, and is typically left out when using software coders.

2.4.1 ITU-T Recommendations H.261 and H.263

ITU-T[8], the Telecommunication Standardization Sector of International Telecommunication Union (ITU), defines two standards (called "recommendations" in ITU-terminology) for transferring video and audio over digital lines. H.261 [20], finished in 1990, is designed for ISDN-lines or other media with transfer rates being multiple of 64 kbit per second. H.263 [21], currently a draft standard, is targeted at lines with lower bitrates.

H.261

H.261 supports two resolutions: Common Interchange Format (CIF) at 352×288 pixels, and Quarter CIF (QCIF) at 176×144 pixels. The luminance color component is coded at these sizes, while the chrominance components are reduced to half the size in both directions.

Frames for the three components are partitioned in *blocks* of 8×8 pixels, each of which are transformed, quantized and Huffman-coded separately. A *macroblock* is defined as four neighboring luminance blocks, and one block from each of the chrominance components, making up a 16×16 sub-image.

Two types of frames are defined, intra coded frames and inter coded frames. Intra coded frames are coded as stand-alone frames, while inter coded frames use prediction errors with respect to the previous frame. The coded blocks of inter coded frames may include motion compensation, in which case a motion vector is associated with each macroblock. The motion vector allows specification of a displacement of up to 15 pixels in all directions. The sender may decide not to send blocks that haven't changed since the previous frame.

H.263

H.263 works much like H.261, but there are several extensions, and some modifications. In addition to the two resolutions defined for H.261, H.263 allows the following: 16CIF at 1408×1152, 4CIF at 704×576, and sub-QCIF at 128×96 pixels.

Extensions to H.261 include "PB-frames mode", where two frames are coded as one unit. The latter frame is coded as an intra frame, while the former frame is coded in inter mode, possibly using bidirectional prediction between the previously seen frame, and the intra coded frame of the same unit.

Another extension is the use of unrestricted motion vectors, where motion vectors are allowed to point outside the frame. Edge pixels are used for prediction of the non-existing pixels. In H.263, motion vectors use half pixel prediction, instead of integer pixel prediction.

[8] ITU-T was until February 1993 known as CCITT.

For the coding step, H.263 allows using arithmetic coding instead of the variable length coding used in H.261.

2.4.2 MPEG

The MPEG (Moving Picture coding Experts Group) standards specify coding of video and audio streams, and how synchronization between them is supposed to be done. At 1.2 Mbits per second, 30 Hz and a resolution of 352×240, the quality of an MPEG stream is comparable to VHS video [22]. The standardization effort was initiated in 1988, run by "Joint ISO/IEC Technical Committee (JTC 1) on Information Technology". The standards are said to be *generic*, in that they specify the format of the compressed stream, rather than the method by which the data are supposed to be coded.

MPEG defines three different types of frames [23], as illustrated in figure 2.6. Note that the standard does not specify the frame type sequence, it is left to the encoding application.

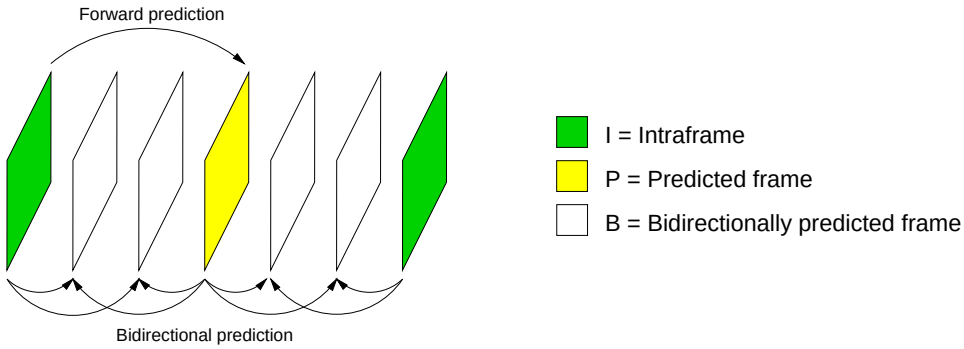

Figure 2.6: The relationship between frame types.

Intraframes, or I-frames, defines the start of a group of frames. I-frames are coded as stand-alone images, using a method resembling the one described for JPEG in section 2.3.1 on page 8.

A group of frames may contain predicted frames, called P-frames. These are predicted from the closest, previous I- or P-frame, with the help of motion compensation vectors. The motion vectors are associated with *macroblocks* of 16×16 pixels.

Between the I- and P-frames, there may be zero or more bidirectionally interpolated frames, or B-frames. These are interpolated between the nearest I- or P-frames. Since the interpolation is bidirectional, the decoder needs to see into the future. Macroblocks within a B-frame can be coded in several ways [22]:

- Intra coding: No motion compensation.
- Forward prediction: The previous I- or P-frame is used as a reference.
- Backward prediction: The next I- or P-frame is used as a reference.
- Bidirectional prediction: Two reference pictures are used, the previous and next I- or P-frame.

Originally, three versions of the standard were planned for different bitrates (1.5, 10 and 40 Mb/s). These were named MPEG-1, -2 and -3 accordingly [24]. Later MPEG-4 was initiated for development, suitable for lower bitrates.

MPEG-1 defines a "Constrained Parameter Set", describing the minimal requirements:

Parameter	Value	Comment
Horizontal resolution	≤ 768	
Vertical resolution	≤ 576	
Macroblocks per frame	≤ 396	$= 288/16 \times 352/16$
Macroblocks per sec.	≤ 396	$= 288/16 \times 352/16$
Frame rate	$\leq 30Hz$	
Interpolated frames	≤ 2	
Bitrate	$\leq 1856kb/s$	

Table 2.1: The Constrained Parameter Set of MPEG-1.

The maximum frame size is 4096×4096.

MPEG-2 offers extended audio-capabilities compared to MPEG-1, including more audio channels, and more sample rates.

MPEG-3 no longer exists. It was developed in parallel with MPEG-2 to support High Definition television (HDTV). As MPEG-2 came to cover what MPEG-3 was supposed to cover, further development was shut down in 1992.

MPEG-4 is the "very low bitrate"-version of MPEG, suitable for bitrates lower than 64 kb/s. It is scheduled to result in a draft specification in 1997 [19].

2.5 Discussion

Table 2.2 illustrates the different sizes of video streams (MPEG) and the corresponding single image streams (JPEG and GIF).

Three MPEG files were recoded to JPEG and GIF[9]; `bart-temple.mpg`, `bjork.mpg` and `enterprise.mpg`. The movies contain 960, 231 and 400 frames respectively, with sizes 192×144, 160×120 and 176×144.

	bits per pixel		
Compression method	`bart-temple`	`bjork`	`enterprise`
None	24	24	24
MPEG	0.84	0.97	0.58
JPEG	1.49	1.64	1.22
GIF	5.37	7.55	5.10

Table 2.2: Sizes of a sample video stream using different types of compression.

Compressing the stream using JPEG requires about twice the bandwidth of the orig-

[9]The recoding process is explained in appendix E on page 119.

inal MPEG stream, while using GIF expands the size to between six and nine times the original, sacrificing most of the colors in the process: GIF supports only 256 colors, while MPEG streams and JPEG images both may contain 16.7 million colors.

Note that the above results should be taken as an illustration of approximate interrelation between results from the different methods. A more serious comparison of the three compression formats should include a measure of distortion from the original images, and it should not use a decoded MPEG stream as the source, but rather the original, uncompressed movies.

2.6 Summary

A video stream consists of bytes representing pixel values. For color movies, each pixel is typically represented by three bytes. A collection of pixels make up a frame, a still image of the scene at a certain time. A sequence of frames make up the video.

Digital images and video are resource demanding when it comes to storage or transfer requirements. It is thus often necessary to compress the data by finding alternate representations. One may take into account the way the human visual system works, and remove certain information without making the loss too noticeable for human spectators.

Single image compression consists of three steps: Transform, quantizing and coding. The transform, typically DCT, reorganizes the pixel data. The quantizer removes "unnecessary" information, while the coding step performs a general compression scheme on the remaining data.

When compressing video, one may take advantage of similarities between nearby frames. With motion compensation, the coder tries to find the most equal block (small sub-image) in an already seen frame, by searching a small neighborhood of the current block. The current block is then coded using the prediction error from the matching block.

Two families of international video compression standards exist: The CCITT family, including H.261 and H.263, and the MPEG family.

Chapter 3

Transferring Video on the Internet

Since the World Wide Web can be seen as a "virtual network" on top of the Internet, making video available on the Web will rely heavily on Internet protocols.

This chapter focuses on ways to transfer video on the Internet in general. It starts with a short introduction to Internet networking, describing the basic protocols from which specialized protocols are built. After that, the use of multicasting for video conferencing is issued. Finally, methods for general data transfer, which may be used for video, are explained, followed by a description of methods designed for video and possibly sound.

3.1 Introduction to TCP/IP Networking

The Internet is a network of computer networks communicating with each other using the TCP/IP protocol suite. Networking protocols are normally developed in *layers*, with each layer responsible for a different part of the communication. A *protocol suite* is a combination of protocols for different layers. TCP/IP is normally divided in the four conceptual layers illustrated in figure 3.1.

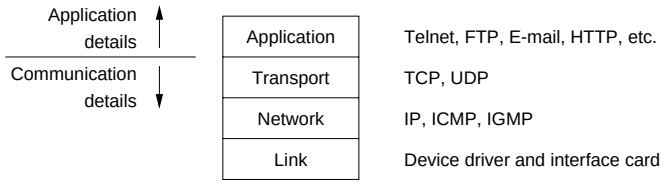

Figure 3.1: The four layers of the TCP/IP protocol suite (From [25, section 1.2]).

The International Organization for Standardization (ISO) has developed a reference model for describing the structure of networks and networking applications, known as the Open Systems Interconnection (OSI) model. This model consists of more layers compared to figure 3.1, but the traditional four layer system should be sufficient to give an overview of TCP/IP networking. For more on the OSI model, see for instance [26].

Data is moved across the network in units called *packets*. Each layer performs *encapsulation* by adding a header and possibly a trailer to the packets. Encapsulation information may include source and destination identification, packet size, checksums, and other controlling information.

The constructed layering offers the benefit of detail hiding: A layer provides a set of well-defined services to the layers above, and relies on the services provided by the layers below.

3.1.1 Link Layer

The link layer includes the networking card and the device driver within the operating system kernel. The responsibility of this layer, is to handle the hardware details. At this level, hosts are identified using addresses stored in the interface card, known as MAC[1]-addresses in the OSI-model [26].

3.1.2 Network Layer

The network layer, sometimes called the internet layer, handles movement of packets around and between networks, including routing. Most network layers have a maximum packet size, based on the characteristics of the underlaying link layer. This is called the network's *maximum transmission unit (MTU)*. When transferring packets exceeding the MTU, *fragmentation* may occur: The packet is split in two or more *fragments*. The destination network layer is responsible for *reassembly* of the fragments into the original packet [1].

IP addresses are introduced at the network layer, as an abstraction from the hardware addresses used at the link layer. The latter are used within a single, physical network only.

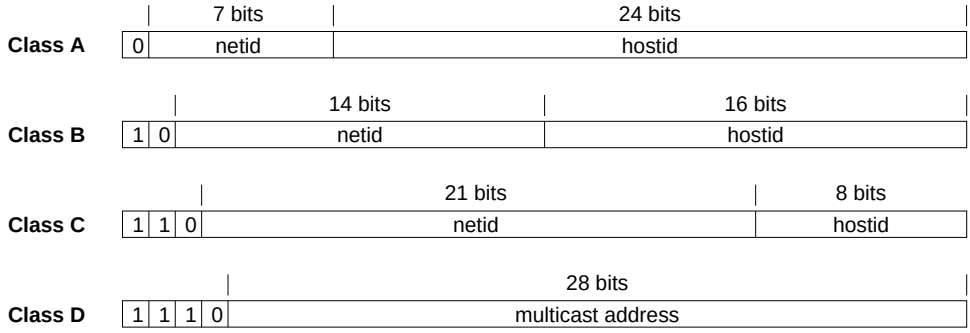

Figure 3.2: IP address classes (From [1, section 5.2.4]).

Every host on the Internet must have a unique 32 bit IP address, encoding a network ID and a host ID. An IP address is typically written in *dotted decimal* notation, where the four bytes of the 32 bit number is written in decimal, separated by dots. To be able to scale for different size networks, the single host address space is divided in the three classes A to C for networks with varying numbers of hosts, according to figure 3.2. In addition, a separate class is defined for multicast addresses, along with

[1]**MAC (Medium Access Control)**: A sub-layer within the data link layer in the OSI model.

a fifth class E (not in the figure) reserved for future use. The host part of the IP address may be split in a subnet ID part, and a host ID part [27]. This *subnetting* eases administration of physically separated networks within an organization.

Humans tend to prefer textual names to IP addresses, so a distributed database, the Domain Name System (DNS) [28][29] exists, mapping between names and addresses.

3.1.3 Transport Layer

The transport layer provides a flow of data between two hosts, to be used at the application layer above it. Two transport protocols exists in the TCP/IP protocol suite:

TCP (Transmission Control Protocol) [30] provides a *connection oriented*, reliable *stream* of data between two hosts. Providing the data as a stream, hides the fact that data is split in packets before being transferred across the network. Making the stream reliable, includes checking that all packets arrive by the help of acknowledgments, timeouts and retransmissions, and assembling them in correct order guided by sequence numbers within the packets. Packets may arrive out of order if the routing mechanism decides to send them through different network paths.

TCP is used by many applications, such as Telnet, Rlogin, FTP and electronic mail (SMTP).

UDP (User Datagram Protocol) [31] on the other hand, just sends packets of data, called *datagrams*, from one host to another. It is up to the application to make sure that packets arrive at the other end, and to sort them in correct order if desirable.

UDP is typically used for applications sending small amounts of independent data, like clock synchronizers and hostname lookup services, and for programs sending packets of full state info, like some networking games.

More than one process on a single host may use TCP or UDP at once. The operating system thus needs a way to identify the source and destination processes of TCP streams and UDP datagrams. A 16 bit *port number*, combined with the protocol type, is used for this identification. Standardized protocols use *well known* port numbers, published in the "Assigned numbers" RFC [32] by Internet Assigned Numbers Authority (IANA)[2]. As an example, a File Transfer Protocol (FTP) [33] client by default connects to TCP port 21 on the server host, since port 21 is the well known port number of FTP.

3.1.4 Application Layer

The application layer handles the application details, aided by the layers below. A class of applications will typically have a commonly defined protocol, describing how they are supposed to communicate. Examples include Simple Mail Transfer Protocol (SMTP) [34], setting a standard for communication between mail transport agents (MTA), and Hypertext Transfer Protocol (HTTP) [35] describing how a Web server and a Web browser does information exchange.

[2] http://www.isi.edu/div7/iana/

For a typical Unix system, the application layer will run as a user process, while the other layers are handled by the operating system kernel.

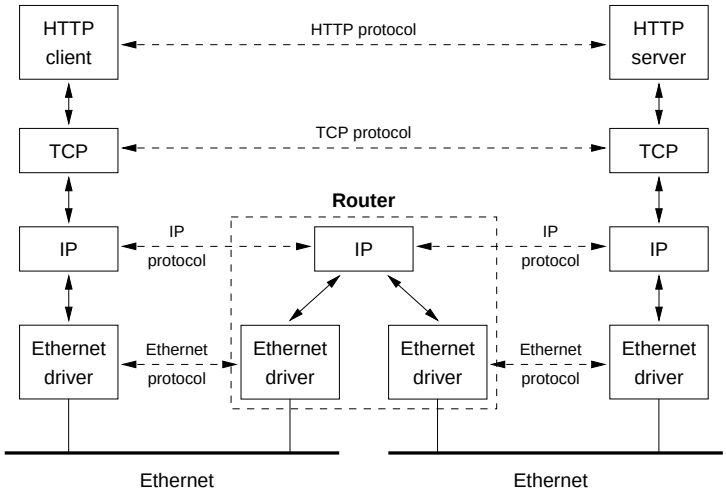

Figure 3.3: HTTP-transfer between two hosts on different physical networks, connected using a router. (Partially from [25, section 1.2]).

Figure 3.3 shows how the different layers of two hosts communicate with each other. Although the data flows through the Ethernet cables, via the router and up or down the layers on each host, we can imagine a *peer-to-peer* connection between the matching layers on the two hosts, illustrated with stippled lines in the figure.

3.1.5 Bandwidth

Bandwidth denotes the data transfer rate of a network line; the number of data units transferred in a given amount of time. The *maximum bandwidth* between two hosts, is determined by the hardware and accompaniment protocols used to connect the hosts in question. If data passes intermediate nodes, the maximum bandwidth is constrained to the one in the bottleneck; the link with lowest maximum bandwidth.

It may be important to distinguish between maximum bandwidth and *available bandwidth*. The available bandwidth depends on the number of connections sharing the same line, routing decisions, and on overhead from higher level protocols. In general, the available bandwidth on the Internet is unpredictable, as lines are shared between many users on different hosts, and TCP/IP doesn't support bandwidth reservations. In addition, routers may choose different paths for the packets comprising a connection.

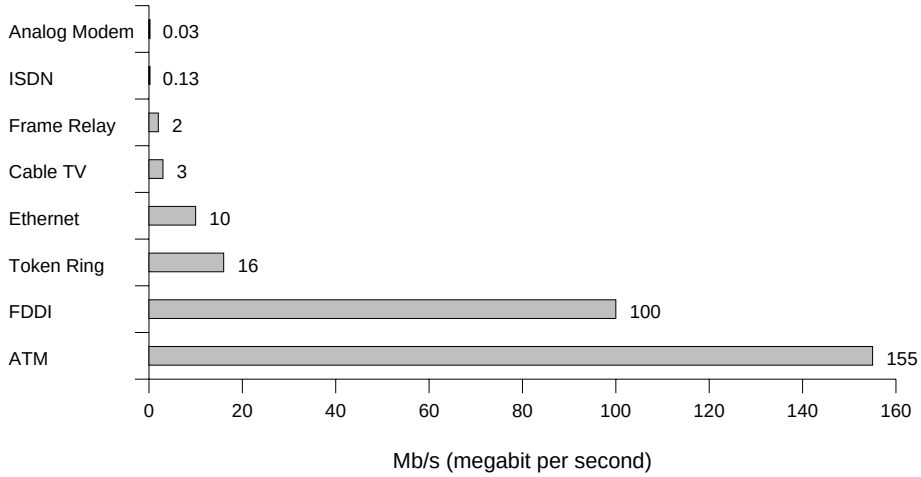

Figure 3.4: Maximum bandwidths for various types of connections to the Internet (From [36]).

Figure 3.4 illustrates the maximum bandwidth on various types of link schemes used to connect nodes on the Internet. At present, analog modem and ISDN (Integrated Services Digital Network) are the most likely connection types for home users. A cable TV provider in Oslo has just recently announced that they will offer Internet connections using their cable network, in cooperation with an ISP. Ethernet, Token Ring and FDDI (Fiber Distributed Data Interface) are LANs, while Frame Relay and ATM (Asynchronous Transfer Mode) are used in Wide Area Networks (WAN).

The problem of varying available bandwidth, plays an important role when transferring real-time video over networks. A decreasing bandwidth may have to be compensated for by transferring less information, doing any combination of the following:

- More extensive compression, typically by increased quantizing.
- Reduce spatial resolution. If displaying at a particular size is important, the receiver may simulate the original resolution by interpolating missing pixels.
- Reduce temporal resolution by lowering the frame rate.

It is important to realize that any reduction in amount of data by the methods mentioned above, will lower the visual quality of the video.

Robust schemes for real-time video should allow a running negotiation between the sender and the receiver about data transfer rate and video quality. The Real-Time Protocol (RTP), introduced in section 3.4.1 on page 26, supports mechanisms for this kind of negotiation.

3.1.6 One-to-many and Many-to-many

In the following, it is important to know that on a Local Area Network (LAN), packets sent may normally be seen by all hosts. Packets not intended for the host in question, are filtered out by the network adaptor, the link layer of the TCP/IP protocol suite.

Traditionally, communication at the application layer of a network has been done

between two hosts only; packets sent have an explicit destination. This one-to-one communication is called *unicast*.

Most LANs also provide some sort of *broadcast*, allowing sending frames[3] simultaneously to *all* hosts on the network. [37] specifies how broadcast is extended to several connected networks on the Internet. Broadcasts are typically used when converting from IP to hardware addresses using ARP (Address Resolution Protocol) [38], or from hardware addresses to IP addresses using RARP (Reverse Address Resolution Protocol) [39].

Modern network interfaces also provide *multicast* [25]. With multicast, packets are accepted by hosts that are members of addressed multicast groups. The filtering of packets is done as a cooperation between the link layer and the IP layer [25]. Multicast is used for delivering packets to multiple destinations in applications for video conferencing and radio and TV transmissions. Bandwidth savings can be achieved compared to unicast, since each package is transmitted only once within LANs. Multicasting on the Internet is described in [40] and [41].

The various cast types are distinguished using hardware addresses on the link layer, and IP addresses on higher layers. Separate sets of IP addresses are used for unicast (classes A to C), multicast (class D) and broadcast. IANA describes the sets and reserves some multicast addresses as "well known addresses" in [32].

3.2 Multicasting and the MBone

Multicast within a single physical network is simple [25]. Problems arise when one wants to use multicast across physical network boundaries. How should a network router decide which packets are to be transported to the outside world, and to what destinations? The Internet Group Management Protocol (IGMP) [42], implemented in the network layer, is used as a solution, aiding hosts and routers in maintaining tables of which hosts belong to which multicast groups. Hosts send IGMP reports when the first process enters a multicast group. Nothing is sent when processes leave a group, but routers send queries periodically, to generate new reports from the hosts.

Hardware routers supporting multicasting are currently not widespread. Instead, software routers, called *tunnels*, are used, encapsulating multicast packets inside regular IP packets. When enabling exchange of multicast packets between two physical networks, a single host on both networks are typically set up to run `mrouted`, the multicast routing daemon. It is expected that commercial routers will support multicast in the near future [43], removing the need for software routers.

A set of multicast capable networks, called *islands*, "connected" using tunneling mechanisms, makes up the MBone (the Multicast Backbone), a "virtual network running on 'top' of the Internet" [44]. The MBone started out as an experiment during the Internet Engineering Task Force (IETF)[4] March-meeting in 1992, located in San Diego. Live audio was sent using multicast transmission to participants at 20 sites

[3] **Frame**: A commonly used name for packets on the lowest level, for instance an Ethernet.
[4] http://www.ietf.org/

on three continents [45]. Over the years, software have evolved enabling other media to be transferred in addition to sound. Today, the MBone is used not only for teleconferencing: In 1993, the Woods Hole Oceanographic Institute used the MBone for transmitting telemetry data from and undersea vessel. Satellite weather photos are transferred as still images, and live activities from space shuttle missions are transferred from NASA's cable TV channel "Select" [46].

3.2.1 Session Management

Video conferences, lectures, and other transmissions of video and sound on the MBone, are often announced to draw attention from the people interested in participating. During the lifetime of a video conference, participants may arrive and leave at various times. A need to invite new participants may also be present. If the conference is encrypted, mechanisms must be available for distributing encryption keys among the participants. Tasks like these are handled using session management protocols.

Session management protocols are currently being specified by the Multiparty Multimedia Session Control (mmusic)[5] workgroup of the IETF. Draft documents are available, and programs implementing the current state of some of the standards exist. The draft standards include:

Session Description Protocol (SDP) [47] defines a session description protocol for advertising multimedia conferences and communicating the conference addresses and conference tool-specific information necessary for participation.

Session Announcement Protocol (SAP) [48] gives description of the issues involved in multicast announcement of session description packets as defined by SDP, and defines a packet format to be used by session directory clients.

Session Invitation Protocol (SIP) [49] specifies how to invite new users to sessions. This is targeted at users who have not joined the conference after seeing it announced using the two above mentioned protocols.

3.2.2 Applications

This section shortly describes a few Unix conference utilities often used on the MBone.

`sdr`, shown in figure 3.5, is a session directory for announcing and scheduling multimedia conferences on the MBone. The program allows users to set up new conferences, or to list and join existing conferences by launching helper applications handling video, sound and shared workspaces. `sdr` uses the draft standards mentioned in section 3.2.1 to perform it's tasks.

[5] http://www.ietf.org/html.charters/mmusic-charter.html

Figure 3.5: `sdr` *main and session information windows.*

The left hand window in figure 3.5 shows the main window of `sdr`, containing the dynamic listing of currently announced sessions on the MBone. The "New" button on the main menu allows the user to announce a new session. When pressing the button, a window (not shown) pops up, asking for information to be broadcasted about the new session.

By clicking on a listed session, the right hand window pops up, giving detailed information on the session in question, including transmission data formats. The window allows users to join the session by launching programs to decode some or all of the transmitted data.

The programs `vic` and `vat`, whose main windows are shown in figure 3.6, may be started either by `sdr`, or as stand-alone programs to handle video and audio conferences respectively.

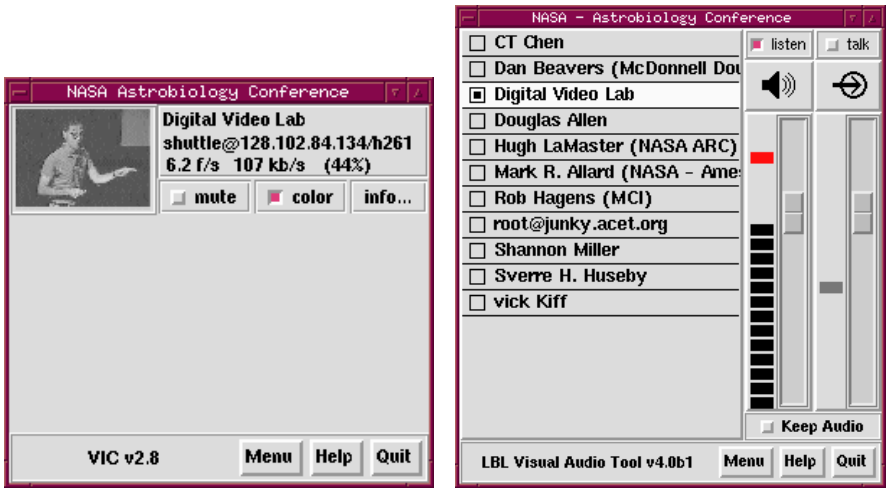

Figure 3.6: `vic` *(left) and* `vat` *main windows.*

The `vic` window to the left in figure 3.6, shows one sender, with the transmitted frames on the left, and information on the right. One may click on the frame window for an enlarged view. The transmitting participants are shown in the main window,

while a list of spectators are available under the "Menu"-button. By default, `vic` doesn't send video until told so by enabling "Transmit" under the "Menu"-button. The program supports various video formats, including H.261 and MPEG.

The right hand `vat` window shows all participants, including those not transmitting. As for `vic`, transmission is off by default. The one currently talking is highlighted.

Shared workspaces or whiteboards, are tools that may be used along with video and audio for video conferencing or lecturing. Figure 3.7 shows the program `wb` in action.

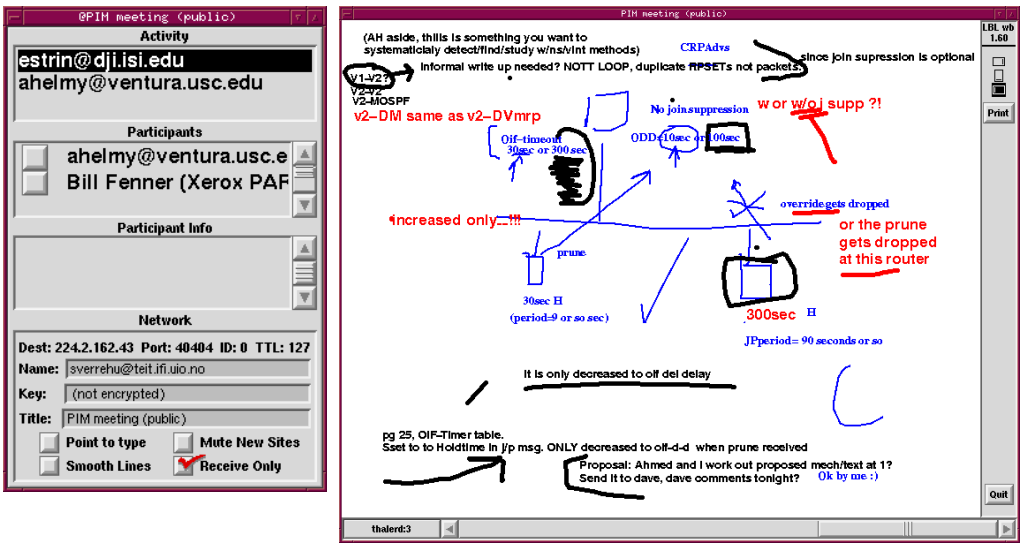

Figure 3.7: `wb` *main and whiteboard windows. (The whiteboard window is slightly shrinked.)*

The main window on the left, shows the current participants. The right window is the actual workspace, where users may write text and draw graphics.

3.3 Methods for General Data Transfer

Methods for transferring unspecified data files, may be usable for various kinds of video applications. When transferring data using a method for general data transfer, the server has no knowledge of the format of the files transferred; there is for instance no way to tell the server to skip a few frames ahead. Possible uses are thus video on demand systems with one of the following playback schemes:

- The entire movie is downloaded before being played.
- The playback speed of the movie is not critical, i.e. no real-time requirements, so playback may be done while the movie is on it's way across the network.

3.3.1 File Transfer Protocol (FTP)

A widely used protocol for transferring files on the Internet, is File Transfer Protocol (FTP) [33]. An FTP client program, normally operated by a human user, connects to a server using TCP. The user may send, receive and delete files, create and remove directories, and perform other file operations across the network. Unrestricted use

of FTP normally requires the user to have an account on the server host. The FTP session is then initiated by the user providing a user name and a password. A popular way of distributing publicly available files on the Internet, is using *anonymous FTP services*, where the user may log in to a public area without having an account on the server host. Users logging in anonymously, are normally restricted to doing downloads only.

3.3.2 Hypertext Transfer Protocol (HTTP)

Even though the WWW is designed to envelope existing protocols, a new protocol was defined for it. The Hypertext Transfer Protocol (HTTP) [35] allows the Web to surmount the problems of different data types using negotiation of data representation [50]. In contrast to FTP, which operates directly on the server file system using file- and directory names, HTTP identifies documents using Uniform Resource Locators (URLs, described in section 4.1 on page 29).

HTTP is a "one-shot" protocol: The client opens a TCP-connection to the server, normally on port 80, and sends it's *request*. The server in turn sends it's *response*, and closes the connection. Several requests to the same server, requires establishing new connections. The repeated reconnectioning that frequently occurs when fetching Web pages, puts an unnecessary load on both the client and the server host, along with the network itself. New versions of HTTP will probably allow a connection to be kept open as long as needed.

The data type negotiation is done using MIME-like headers in both the request and the response (MIME [51] is briefly described in section 4.2 on page 30).

Although mainly being used for transferring data from the server to the client by request, the HTTP standard also defines methods for sending data to the server, used for instance in fill-out forms embedded in HTML-documents. Fill-out forms are handled by special programs running on the server side, communicating with the HTTP-server using the Common Gateway Interface (CGI) protocol [52].

3.4 Methods Related to Video Transfer

When live, or other real-time play is required, the client and server need to negotiate the size of the data transferred, and thus the quality of the movie, to cope with variations in available bandwidth on the network. Several ad hoc solutions are implemented in various programs, but standards are beginning to emerge on the Internet, most of them currently as drafts.

3.4.1 Real-Time Protocol (RTP)

RTP [53] defines functionality for use in applications transmitting real-time data, such as audio and video, over multicast or unicast network services. The functionality includes identification of media type, sequence numbering and timestamping. The data transfer may be aided by a control protocol (RTCP), providing data delivery monitoring, and participant identification for on-going sessions. RTP and RTCP are

typically run on top of UDP, but other transport protocols, such as TCP, may also be used.

Resource reservation and quality of service are not addressed by RTP, but are left to lower layers. Likewise, RTP does not guarantee delivery or prevent out of order delivery, but the sequence number provided by RTP allows the receiver to reconstruct the sending order.

RTP is considered a framework for new protocols, and is thus not directly usable. A header template is defined, but the format of the data to be transferred, the *payload*, is undefined. Application developers will have to create *profile specifications* and *payload format specifications* extending RTP to cope with the medium in question. A profile specification defines payload type codes, and any extensions or modifications to the original RTP. Profiles for audio and video are defined in [54]. The payload format specification defines how the payload, in our case the video data, is to be carried in RTP. Currently, payload formats for MPEG [55], H.261 [56] and JPEG [57] are defined, while others are being developed.

3.4.2 CU-SeeMe

CU-SeeMe is a software package featuring it's own, proprietary, and partly undocumented[6], compression scheme. The package may be used for video telephony and conferencing on Macintoshes and PC's, and has gained some popularity, since the data transfer rate is suitable for modern modems, making the program usable for most people with an Internet connection. The package was originally developed at Cornell University, but a commercial version[7] is also available.

In [58] Tim Dorcey, one of the developers, gives a quick overview of how CU-SeeMe works: A frame is resampled to 160×120 pixels, with each pixel quantized to 16 levels of gray. Following that, the frame is subdivided in blocks of 8×8 pixels. A block is marked for transmission if it differs sufficiently from the previous transmitted block at that location. The difference is measured as the sum of the absolute values of all 64 differences, with an extra multiplicative penalty for differences in nearby pixels.

Before transmitting a block, it is compressed using a simple, ad hoc reversible compression scheme developed by the program authors. The goal of the scheme is to be able to compress and decompress fast. To cite Tim Dorcey, *"What it lacks in mathematical elegance, it makes up for in quickness"*. Compression builds on the assumption that a row inside a block is often similar to the row above it. A 32 bit word is created by combining the pixel values in a row, and the difference with the above 32 bit word is coded using 4, 12, 20 or 36 bits, including 4 bits giving further coding details. The compression scheme is said to reduce the amount of data to transfer by about 40%.

The program uses UDP at port 7648 for transferring image frames between two participants [59].

[6] According to Tim Dorcey, CU-SeeMe is only documented by source code.
[7] http://goliath.wpine.com/cu-seeme.html

In it's original form, CU-SeeMe can be used for one-to-one communication only. Using *reflectors* however, the usability may be extended to real, multi-participant video conferencing. A reflector is a specialized program running on a Unix host, capable of multicasting CU-SeeMe packets.

3.5 Summary

The TCP/IP protocol suite, which is used for communication on the Internet, contains four abstraction layers: The hardware link layer, the routing network layer, the data flow handling transport layer, and the program specific application layer.

Variations in available bandwidth between two communicating hosts on a network, plays a role when transferring real-time information. The information quality may have to be adjusted according to the available data transfer rate.

Packets sent may be intended for a single recipient (unicast), or several recipients (multicast). Using multicast instead of sending the same packets to several hosts with unicasting, may save bandwidth. Multicasting between physically separated networks require special routers, most of which are currently implemented in software. The MBone is a multicasting network on top of the Internet.

Video may be transferred using general data transfer protocols, such as FTP and HTTP. To be able to play real-time while transferring, one needs protocols capable of adjusting the data stream according to the available bandwidth. Most existing protocols are currently not fully standardized.

Chapter 4

Solutions for Embedding Video in WWW Browsers

Berner-Lee's proposal [6] for the project that resulted in the World Wide Web, describes two important building blocks of the Web, the *browser* and the *server*. The browser is the program operated by the user. It's job is to display whatever documents the user requests, in a format suitable for the machine configuration. The server is the information storer and provider, delivering the documents requested by the browser.

One of the basic goals of the World Wide Web was to provide hypertext documents, enabling users to follow links to other documents on the Web. A suitable format was defined, called Hypertext Markup Language (HTML) [60]. The format is evolving to adapt to users' needs, so a version 3.2 is under development by World Wide Web Consortium[1] in cooperation with browser vendors [61].

HTML describes the logical structure of a document rather than it's formatting. This allows different platforms and programs to display the contents according to their own conventions, or the user's preferences.

This chapter discusses various ways of including video in Web browsers. The first section describes URLs, the addresses for documents on the Web. Following that is a section on how Web browsers identify the content types of documents. The sections describing viewing video from browsers, include executing external applications, server push and client pull, animated GIFs, browser source code modifications, plug-ins, and Java programming.

4.1 Uniform Resource Locators (URLs)

On the Web, documents are identified using Uniform Resource Locators, or URLs [62]. The fields of a URL describe the protocol, called *scheme* in URL terminology, used to retrieve the document, in combination with a protocol specific part. For most protocols, the specific part denote the host on which the document may be found, an optional network port to connect to, and a path identifying the document.

[1] http://www.w3.org/pub/WWW/

What follows is a couple of sample URLs:

`http://www.whitehouse.gov:80/WH/Welcome.html`

The above URL indicates a server on the host `www.whitehouse.gov`, listening on port `80`, and communicating using HTTP. The document is further identified with the path `WH/Welcome.html`, indicating that this is a hypertext document. According to [32], port 80 is the default for HTTP, so the `:80` could have been omitted.

`ftp://sunsite.unc.edu/pub/Linux/`

This URL identifies the directory `pub/Linux/` on the server `sunsite.unc.edu`, accessed using FTP [33].

`mailto:sverrehu@ifi.uio.no`

URLs need not necessarily reference existing documents, as illustrated by the above example. The `mailto`-scheme specifies an electronic mail address. Most browsers will pop up a window in which the user may compose an E-mail to the address given.

4.2 Browsers and Document Types

As mentioned above, the browser is responsible for fetching and displaying documents from servers, as requested by users. Most browsers have built-in support for communicating with various types of information providers in addition to HTTP-talking Web servers. The built-in support typically includes FTP (general file transfer [33]), NNTP (news [63]), SMTP (mail [34]) and Gopher (a distributed document search and retrieval protocol [64]).

Before displaying a document, the browser needs to decide the contents, to choose an appropriate way of viewing. Documents are typically categorized using MIME-like descriptors. MIME [51] is an Internet standard for identifying the contents of mail, using a classification scheme built on types and subtypes. As an example, an MPEG video stream would be classified as `video/mpeg`.

When transferring a document using HTTP, the HTTP header may, depending on the server, include a `Content-Type` field giving the type and subtype of the document in question. If no `Content-Type` field is present, or if another protocol than HTTP is used to transfer the document, the browser will have to do a qualified guess on the contents. This guessing is typically done by examining parts of the URL used to reference the document, in particular by looking at the extension of the filename part of the URL. Browsers may be configured to associate certain extensions with MIME-types.

Once the type of the document is identified, the browser decides how to display the contents. Widely used formats, such as HTML, plain text, GIF and JPEG, are normally handled by the browser itself. Other formats may be sent to external applications or plug-ins, as described below. Handling of initially unsupported MIME-types, may normally be specified by the user.

4.3 Spawning External Applications

The simplest way to make a Web browser handle documents of types it normally doesn't know of, is to make it start an external program to view the document.

There are a couple of drawbacks to this: First, the feeling of integration is lacking. The helper program shows up in a window of it's own, making it difficult to combine the document contents with other information. Also, the look and feel of the separate application may not match that of the browser. Secondly, the user will have to install the helper application, including setting up the browser to call it for appropriate document types. For novice users, this may incur problems, as may the task of identifying and retrieving the application.

The problem of integration may be solved to a certain extent by specifying protocols for how external applications may use a subwindow of the browser. Ideally, the protocol should include some negotiation of the proper window size.

The benefit of using external helper applications, is that one may use highly specialized, already existing programs.

The rest of this chapter will discuss methods for integrating video within the browser window.

4.4 Server Push and Client Pull

Early methods for displaying "animation" in Web pages, include *server push* and *client pull*, both developed by Netscape Communications Corporation[2]. Server push and client pull are described in [65].

When using server push, the connection between the browser and the server is kept open. The server, or a CGI-program running at the server side, splits data in chunks, and sends the chunks one by one in a multi-part MIME-message. The browser handles each part as it arrives.

Client pull works by forcing the browser to reload a document, or load a new document in a given amount of seconds. The delay is passed to the browser using the HTTP response from the server, or as a special `META`-tag within the HTML-document being passed.

To transfer video using any of these methods, the server will either have to decode the video stream, and recode it as single images in a format the Web browser in the other end can handle, or store the images pre-encoded.

There are several drawbacks making these methods inappropriate for transferring video on the World Wide Web: First of all, they are not bandwidth friendly. According to table 2.2 in section 2.5 on page 14, GIF and JPEG requires several times the bandwidth of the original MPEG stream.

In addition to requiring a high bandwidth, and thus putting a big load on the network, the process of decoding and recoding images may take up most of the CPU time of the server. If several people request movies at the same time, the server will have a hard time serving them all. Decoding is better left off at the client side.

[2] http://home.netscape.com/

The timing resolution using client pull is one second; the "next" image may be loaded in any number of whole seconds, or immediately. For real-time video, this is not accurate enough.

A semester project at the University of Illinois at Urbana-Champaign included testing server push and client pull of MPEG streams recoded to GIF images [66]. The results demonstrate that server push and client pull are not usable for transferring video.

Even if inappropriate for large scale video, server push and client pull have been used extensively for highlighting Web pages by adding small, animated widgets drawing the spectator's attention. However, using server push or client pull for this purpose, is currently being overtaken by animated GIFs.

4.5 Animated GIFs

Compuserve's second version of GIF, known as GIF 89a [14], allows timed delays to be inserted between images in a multi-image GIF file, making it possible to use GIF for animations. The timing resolution is 1/100 second. One may also specify a count of times to rerun the animation, or make it run infinitely.

Animated GIFs have gained popularity recently, being used for spicing up Web pages, especially in commercial advertising. Several freely available tools for making animated GIFs from a sequence of single images has become available.

Compared to client pull, that was formerly used for the same purposes, animated GIFs put less load on the network, since images are transferred only once. Since the entire file is sent to the client, it may be cached locally, resulting in faster play, and faster reload of the page including the animation.

4.6 Extending Browser Source Code

The most direct way to include video in a browser, is to extend an existing browser by modifying it's source code. For this to be possible, the browser source code must be available, as is the case with for instance NCSA Mosaic[3], the first widely known browser.

[67] describes rewriting Mosaic to support real-time video and audio, to a browser called Vosaic[4]. Vosaic accepts URLs of the form

```
mbone://224.2.252.51:4739:127:nv
```

referencing an MBone connection at address 224.2.252.51 at port 4739, having a Time To Live (TTL) of 127, and using the `nv` transmission format.

Rewriting browser source code requires people to install the modified version of the browser, which is not necessarily the browser of choice. In addition, when programmers around the world create their own versions of browsers, each extended to do various useful things, it will be difficult to incorporate all wanted behavior into a

[3] http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/
[4] http://choices.cs.uiuc.edu/research/Vosaic/vosaic2.html

single browser. This leads to a scenario in which people use different browsers for different tasks, breaking with one of the basic goals of the WWW philosophy; incorporating all information into a single client program. Later sections describe ways to extend browsers in somewhat standardized ways, making it possible to add independent extensions to the same browser program without recompilations.

4.7 Plug-ins

Some browsers allow their functionality to be extended by third party *plug-ins*. A plug-in is a dynamically loadable library specialized in interpreting and presenting a certain data format. The plug-in will run as part of the browser process, and have access to limited areas of the browser window, thus appearing to the user as part of the browser.

A browser decides what plug-in to hand a document to by examining the extension of the document "filename". Extensions are bound to MIME-types describing the expected data type in the document, and documents are handed to a plug-in that has announced it can handle the MIME-type in question. The document retrieval is done by the browser, so plug-ins may currently not be used to handle data sent using protocols not supported by the browser. This limitation, which plays a role when real-time negotiation is required between the plug-in and the server providing the data, may be circumvented by letting the initial document contain information necessary for the plug-in to set up it's own connection. Future browsers should let plug-ins not only specify what MIME-types they expect, but also protocols provided.

As of 1996-10-11, BrowserWatch [68] listed 53 known WWW browsers, of which four was indicated as having plug-in support. Netscape Navigator, the most widely used browser [68], was the first to allow third party plug-ins. Fortunately, the other three browser developers have decided to stick to Netscape's plug-in format[5], making available plug-ins sharable.

Like is the case with external applications, plug-ins will have to be installed before being used. Netscape gives guidelines for how plug-in developers may simplify this process for the user, and for how HTML documents requiring plug-ins for embedding automatically may let the user download the plug-in if not already available [69].

In Netscape Navigator, plug-ins may be either *embedded* or *full screen*. An embedded plug-in shows up along with other information elements, and require an `EMBED`-tag inside a HTML-document. Figure 4.1 gives an example of a HTML-document with an `EMBED`-tag. A plug-in is activated as full screen when a document with the MIME-type of the plug-in is accessed directly. Full page plug-ins are typically used for document viewers.

[5] More on Netscape plug-in API in section 5.1 on page 41

```
<HTML>
<H1>The MPEG plugin</H1>
Displaying <A HREF="bart-temple.mpg">bart-temple.mpg</A> after loading.
<P>
<EMBED
    SRC="bart-temple.mpg"
    WIDTH=300 HEIGHT=200
    TransferMode=LocalFile>
</HTML>
```

Figure 4.1: Example of an `EMBED`*-tag in a HTML-document.*

Note the use of `WIDTH` and `HEIGHT` to set the size of the window in figure 4.1. Setting the size of embedded windows is required in Netscape Navigator. The "correct" size of the window is often not known until parsing of the video stream has started, but Navigator doesn't let a plug-in resize the window. For the convenience of both users and programmers, future browsers could allow resizing of embedded windows, followed by restructuring of the surrounding document contents.

Figure 4.2: A plug-in viewing MPEG movies in Netscape Navigator. On the left, the plug-in is embedded in a HTML-document, while the right image shows a full screen plug-in.

Figure 4.2 shows the difference between an embedded and a full screen plug-in. Note that other text of the HTML-document surrounds the embedded plug-in, which is shown in a sub-window. The full screen plug-in occupies the entire view area of the browser window.

The Plug-in Plaza of BrowserWatch [68], striving to list existing plug-ins, shows several plug-ins for displaying video inside a browser. The plug-ins may be used to

view MPEG or QuickTime movies on Macintosh and MS Windows based computers. Plug-in Plaza lists no plug-ins capable of inline video conferencing or other two-way communications.

Chapter 5 on page 41 describes the implementation of a Unix-plug-in capable of displaying MPEG-movies inside Netscape Navigator, shown in figure 4.2.

4.8 Java Applets

The origins of Java go back to 1990, when Patrick Naughton announced that he would leave Sun Microsystems in favor of NeXT Computer Inc. To keep him from leaving, Sun gave him permission to work on anything he would like, with some of the best programmers available at the site, one of which was James Gosling. The initial goal of the project, was to create a device that could control everyday consumer appliances through an animated, graphical interface. Gosling's job was to define and implement a programming language suitable for the task. The programming language was named "Oak".

When the device prototype was finally ready, Sun failed to convince potential customers to incorporate the device in their products. In 1994, the project was closed. It was then Bill Joy, a Sun co-founder, got the idea of bringing the programming language to the Internet, by letting it extend the capabilities of the World Wide Web. The language itself became the product, instead of just part of a device. Gosling adapted the language, which was now renamed "Java"[6], to the Internet, and Naughton wrote "HotJava", a Web browser supporting the language. In an attempt to get a hold on the Internet market, Sun decided to give the products away for free. It has later turned out to be a success [70][71].

4.8.1 What is Java?

Java is an object oriented programming language, highly inspired by C++. What separates it from other popular programming languages, is that a program is normally not compiled to machine code for a particular processor and operating system type, but rather to a binary code interpreted by a "Java Virtual Machine" (JVM) [72]. This way, a precompiled Java program may be distributed and run on any computer or other device where a Java Virtual Machine is available. Currently, JVMs are typically available as programs for various platforms, but Sun recently announced *picoJava*, a microprocessor that executes Java byte code directly.

A Java program may be run either as a stand-alone application, or as an *applet* loaded as Java byte code from some host on the Internet by a Web browser. Applets are programs run by Java capable Web browsers, showing their output and accepting user input from within the browser window. In effect, applets bring full interactivity to the World Wide Web: An information provider may write a Java program that visualizes the information in a suitable way, possibly guided by user input. The applet concept is a major reason for Java's popularity.

Using interpreted code may slow down execution compared to running pure machine

[6] http://www.javasoft.com/

code. Java is normally shipped with an extensive class library, the Java Application Programmer's Interface (API) [73], compiled in the host's native language. The class library handles a great range of tasks, such as file and network I/O, user interfaces and windowing, and image processing. If a Java program fully utilizes the library, most of the execution time will be spent in the library code, thus minimizing the loss of speed introduced by using interpreted code.

A programmer may combine Java code with methods[7] and libraries written in C, and compiled to machine language. Such methods and libraries are called *native* [74]. Native code may be used for speed critical parts of a program, but it's use breaks one of the main reasons to use Java in the first place; portability. In the future, JVMs will probably translate the Java byte code to machine code, either before or while running the program. Translating the entire program to machine code, will eliminate the need for using native code to gain speed.

Java applets are embedded within HTML-documents using an `APPLET`-tag. Figure 4.3 gives a sample of an included Java applet.

```
<HTML>
<H1>Java Video Applet</H1>
<APPLET CODE="SHHVid.class" WIDTH=160 HEIGHT=120>
    <PARAM NAME="port" VALUE=8195>
    You're supposed to see an applet here.
</APPLET>
</HTML>
```

Figure 4.3: Example of an `APPLET`*-tag in a HTML-document.*

As for plug-ins and `EMBED`-tags, `WIDTH` and `HEIGHT` -parameters are required for `APPLET`-tags, setting the size of the subwindow in which the applet runs. Note the use of a `PARAM`-tag in figure 4.3. Such tags are used to pass parameters to the applet. Applet programmers specifies a set of parameters accepted by the applet, to let users perform appropriate initialization. The "You're supposed to see an applet here"-text is displayed by browsers not supporting the `APPLET`-tag.

[7] In object oriented programming languages, *methods* are functions defined within a class.

Figure 4.4: A sample Java applet displaying live video, run within Netscape Navigator.

Figure 4.4 shows an applet in action, embedded in a HTML-page viewed using the browser Netscape Navigator. This real-time applet displays images received from a remote camera. It's implementation is described in chapter 6 on page 55.

4.8.2 Java Applets and Security

Downloading and automatically executing programs from anywhere on the Internet, may be a major threat to the security and integrity of the local system. Java applets are thus denied access to the local filesystem, or given access to a separate part where no harm can be done. When creating network connections, applets are only allowed to connect to the local host, or the remote host from which the applet was loaded.

The security restrictions are handled by a special `SecurityManager`-class, normally subclassed by the browser. Before doing any possibly restricted actions, the methods in the Java class library call the security manager, telling it about the intended action. If the action is not allowed, the `SecurityManager` throws a `SecurityException`, aborting the applet [73].

The prohibition to connect to any host on the Internet except the local host or the host from which the applet was loaded, may be troublesome when using Java applets for bringing video to Web pages. The remote Web server may not be the source of the video. There are a couple of ways to circumvent the restriction:

1. Set up a Web server on the host providing the video, and let this server send the applet.
2. Use a *proxy* (a program operating on behalf of another) on the Web server host, and let this program redirect the communication to the host providing the video.

The first alternative may be the simples to implement, but may not always be possible, as the author of the Java applet may not have access to the video providing host.

Chapter 6 gives an example of using a proxy as a gateway between a camera grab server and a Java applet.

4.8.3 Using Java for Video

At present, no classes in the Java API handles coding or decoding of image streams. There are no classes for discrete cosine transform, Huffman coding, bitstreams and other tools for compression, so everything must be built from scratch. Likewise, there is no support for fetching images from a camera connected to the computer. Programming support for cameras using Java only may be tricky, if not impossible, since access to hardware, special code libraries or a server program may be required.

At present, the solution must be to extend the Java library using native code, or to let Java programs talk to external programs providing the lacking functionality. In the future, video handling and communication with camera equipment may be part of the ever growing standard Java library.

4.9 Discussion

Comparing external applications to plug-ins, we see that plug-ins solve the problem of integration. To a user, the plug-in will appear as just a part of the browser. As will be demonstrated later, it is possible, maybe with a little hassle, to create a plug-in from an already existing application.

When comparing writing plug-ins to rewriting browser source code, it is clear that plug-ins makes it possible to easily install several extensions to the same browser, without requiring any programming knowledge from the user. As more browsers support the same plug-in API, the user may depend on the browser of choice, instead of switching to a different browser to get the wanted behavior. Rewriting the browser gives one benefit over plug-ins: The programmer may incorporate features that are not (currently) available using the plug-in API, such as directly handling protocols not supported by the browser.

Java offers platform independency compared to plug-ins. In addition, a Java applet is downloaded and compiled automatically at runtime, thus freeing the users from doing manual installations. Also, the automatic download will guarantee that users always run the latest, and hopefully better, release of the viewer. However, as Java programs at the moment are run by interpreters, Java scores somewhat low when it comes to speed of non-trivial programs, compared to plug-ins and external applications. As on-the-fly Java byte code to machine code translators, and hardware Java Virtual Machines become widely available, slowness will no longer be a reason to stay away from Java.

4.10 Summary

On the Web, documents are identified using URLs, describing the transfer protocol, and often the server host and file location. When viewing a document, the browser identifies the document type either by examining parts of the URL, or, if the transfer protocol is HTTP, by looking at headers sent by the HTTP-server.

There are several ways to make a Web browser handle documents of which it has no built in knowledge, in our case video. The browser may start a helper application, an external program written to handle the video. For simple animations, server push, client pull or animated GIFs may be used.

Full integration of video may be achieved in several way, including the brute force method of rewriting the browser source code. Rewriting the source code makes it hard to incorporate several extensions from various developers. Plug-ins allow browser code extension without changing the code, using dynamically loaded libraries. Using plug-ins, developers may distribute platform dependent, precompiled extensions to several popular browser programs. Java applets are programs written in the Java language, interpreted by Java Virtual Machines in several of the major browsers. The main benefit of Java is portability, as Java programs are not compiled to machine code, but rather to Java byte code. Currently, Java has little built-in support for video, but the future will hopefully change this.

Chapter 5

MPEG Plug-in for Netscape Navigator

The most common way to view MPEG-movies from a browser on the Unix platform, is to spawn an external application. This chapter describes the work spent on programming an MPEG plug-in for Netscape Navigator, one of the most popular browsers for the World Wide Web. The plug-in is implemented for Unix platforms running the X11 Window System. Figure 4.2 on page 34 shows the plug-in in action.

The following sections will try to explain what a Netscape plug-in is, and briefly, how to program one. The focus is then moved to the implementation of the plug-in, including choosing an available MPEG decoder, and converting it into a library suitable for use by the plug-in. The problems of multiprocess decoding, and viewing colors under X11, are shortly issued.

5.1 Netscape Plug-in API

Netscape allows programmers to enhance the Navigator[1] functionality by creating dynamically loadable libraries, called plug-ins. When the Navigator starts up, it queries available plug-ins for MIME-types [51] they can handle. Whenever a document with any of those MIME-types is opened in the browser, the appropriate plug-in is loaded. Navigator passes the document to the plug-in either as a stream while loading, or as a file after loading. Whichever method is used, is controlled by the plug-in. Netscape encourages plug-in developers to use the streamed transfer mode whenever possible. Downloading to a local file before viewing, may take some time for larger documents.

Netscape defines a Plug-in Application Programmer's Interface (API) [69] giving guidelines for plug-in developers. When calling a plug-in, Navigator *expects* certain functions to be defined, called "plug-in methods". Also, Navigator *provides* a set of functions for the plug-in to call, termed "Netscape methods". Functions defined by the plug-in have names starting with `NPP`, while function names provided by Navigator start with `NPN`.

What follows is a short description of the most important of the above mentioned

[1] http://merchant.netscape.com/netstore/navigators/

functions. Let's start with the plug-in methods. The order of the functions resembles the calling order.

The following functions are used when the plug-in is initiated.

`NPP_Initialize` is called when the plug-in is loaded, before any streams are handed to it. The function is supposed to do general plug-in initialization.

`NPP_GetMIMEDescription` is used by Navigator to query the MIME-types this plug-in supports.

For each new instance of the plug-in, Navigator calls the following functions:

`NPP_New` Called when a new instance of the plug-in is wanted. This is typically when the user enters a page containing data this plug-in gives support for. Among the parameters to this function, is a pointer to an instance specific structure. The plug-in should allocate and initialize a piece of memory to hold any instance data required, and enter a pointer to this memory into the instance data structure. The instance data structure is passed in every call to functions handling an instance.

`NPP_SetWindow` is used to give the plug-in a sub-window of the browser, or to indicate that the window size has changed. The first time this function is called, the plug-in normally sets up the layout of its window.

There are several functions used for stream handling:

`NPP_NewStream` indicates that a new stream is available. By returning certain values, the plug-in may request that data is passed either as a file, or in a streamed manner.

`NPP_StreamAsFile` passes a filename to the plug-in. This function is called when the plug-in has requested the data as a file. The plug-in may open and process the file as it wishes.

`NPP_WriteReady` is used when data is transferred as a stream. Netscape Navigator calls `NPP_WriteReady` whenever it has something to send to the plug-in, to query the number of bytes it is able to handle. The number returned indicates a promise rather than a limit; Navigator may send more (or less) bytes than accepted by the plug-in, but the plug-in doesn't have to read more than returned by the last call to `NPP_WriteReady`. The actual data transfer is done in `NPP_Write`, which is called afterwards.

`NPP_Write` lets Navigator transfer a block of bytes from the stream to the plug-in. The plug-in is supposed to accept at least as many bytes as it promised in the last call to `NPP_WriteReady`.

`NPP_DestroyStream` is called to indicate that there are no more data available in the stream. Among the possible reasons passed as a parameter, are: End of stream reached, user break and network error. Note that the destruction of the stream does not imply destruction of the instance. The instance is still supposed to be around to repaint the window.

Finally, functions are called to tidy up when an instance, or the entire plug-in is no longer needed.

`NPP_Destroy` is called when an instance is no longer needed. This is typically when the user leaves a page containing the window handled by this instance of the plug-in.

`NPP_Shutdown` is the opposite of `NPP_Initialize`. It is called just before the plug-in is unloaded.

Most functions provided by Navigator is of no interest to the MPEG plug-in, so only a few are mentioned:

`NPN_MemAlloc` may be used by plug-ins to allocate memory from Navigator's memory space.

`NPN_MemFree` frees memory returned by `NPN_MemAlloc`.

`NPN_GetURL` requests that Navigator opens a new URL. The resulting stream may be handled by Navigator itself, or by the plug-in, depending on parameters passed.

`NPN_Status` enables displaying messages on Navigator's status line.

5.2 Choosing an MPEG Decoder

To save a lot of time, an already existing MPEG decoder is used as a basis for the plug-in. In deciding which decoder to use, the following properties were sought:

- Freely available source code.
- Freely redistributable, both in source and binary-only form, to let others continue working on the plug-in, possibly to make it fully usable.
- Tidy, commented and easily changed/ported code, since the code will have to be changed to adapt it to the plug-in.
- Somewhat optimized for speed.

Two alternatives were found, both mentioned in the MPEG-FAQ [19].

5.2.1 mpeg_play-2.3-patched

This source code package[2] originated at University of California, Berkeley. It was written by Lawrence A. Rowe, Ketan Patel, Brian Smith, Steve Smoot, and Eugene Hung. According to the `README`-file, it implements the MPEG standard described in the Committee Draft ISO/IEC CD 11172 dated December 6, 1991 which is sometimes referred to as "Paris Format". There is no support for real-time synchronization or audio.

The decoder will display the movie in an X window on an 8, 24 or 32 bit display. When counting variations over the same scheme, 18 dithering algorithms are implemented, including Floyd-Steinberg, ordered dither, and halftoning.

[2] ftp://mm-ftp.cs.berkeley.edu/pub/multimedia/mpeg/play/

The source for this program contains 19 910 lines in 33 files named `*.[ch]`. It is sparsely commented.

The program may be used, copied, modified and distributed freely, as long as a copyright notice is not removed.

5.2.2 mpeg2play-1.1b

This program[3] was written by Stefan Eckart, based on `mpeg2decode` by MPEG Software Simulation Group. It is a player for MPEG-1 and MPEG-2 video-streams, with no support for audio.

The video sequence is displayed in an X window on an 8 bit display, using ordered dither.

Number of lines of code in `*.[ch]` (15 files) sums up to 7 355 for this program. The comment-to-code ratio is about the same as for `mpeg_play`, that is; the code is sparsely commented.

The distribution policy of this program is not known, as the author did not answer a mail regarding this question.

5.2.3 Benchmarks

The programs are compiled using `gcc`, optimized using `-O2`. Testing is performed on a Silicon Graphics Indy, powered by a 100 MHz MIPS R4600 with 16 kb data- and instruction cache, and 32 Mb RAM. The movie-files were stored on a local disk, to avoid letting NFS slow things down. There were no other users on the computer.

Timing is done using GNU's `time`-program. The numbers given are sums of user and system time for the process in question, averaged from three runs with no major page faults.

The Testfiles

Info on the MPEG-files used for testing, are collected using `mpegstat`[4]. Table 5.1 gives some characteristics for the files. Sizes are given in kilobytes (kb).

File	Size (kb)	Frames	Dimen	Sequence
`enterprise.mpg`	723	400	176×144	IBBPBB
`bart-temple.mpg`	2 706	960	192×144	IBBPBB
`bjork.mpg`	523	231	160×120	IBBPBB

Table 5.1: MPEG-files used for testing.

`enterprise.mpg` Raytraced movie of "Starship Enterprise" flying into sunset. Small movements, smoothly scrolling background.

[3] ftp://ftp.netcom.com/pub/cf/cfogg/mpeg2/
[4] ftp://ftp.crs4.it/mpeg/programs/

`bart-temple.mpg` A cut from "The Simpsons". Contains both static and rapidly changing backgrounds.

`bjork.mpg` From Björk's music video "Human Behavior". Lots of rapidly changing scenes.

mpeg_play-2.3

The first test examines the speed of MPEG decoding only, skipping dithering and X11 display output. The following command was used:

```
mpeg_play -framerate 0 -no_display -dither none -quiet file.mpg
```

The results are as follows from table 5.2.

File	Time (sec)	FPS	RAM (kb)
`enterprise.mpg`	6.28	63.7	3808
`bart-temple.mpg`	23.13	41.5	3840
`bjork.mpg`	4.34	53.2	3712

Table 5.2: `mpeg_play` *with no display and no dithering.*

The following tests regard the speed of miscellaneous interesting dither options. First the `ordered2` dither, which is the default dither for `mpeg_play`. `ordered2` dithers to eight bitplanes. The following command was used:

```
mpeg_play -framerate 0 -no_display -dither ordered2 -quiet file.mpg
```

Table 5.3 shows the performance of decoding and dithering using `ordered2` dither. Compared to table 5.2, the dithering requires approximately 30% of the time spent.

File	Time (sec)	FPS	RAM (kb)
`enterprise.mpg`	9.09	44.0	4560
`bart-temple.mpg`	33.38	28.8	4640
`bjork.mpg`	5.77	40.0	4432

Table 5.3: `mpeg_play` *with no display and* `ordered2` *dither.*

Full color dither may be used on displays with more than eight bitplanes, and will thus give high quality results. For testing this dither, the following command was issued:

```
mpeg_play -framerate 0 -no_display -dither color -quiet file.mpg
```

As table 5.4 shows, full color dither is slower than `ordered2`, taking approximately 40% of the total decoding time.

File	Time (sec)	FPS	RAM (kb)
`enterprise.mpg`	11.46	34.9	3952
`bart-temple.mpg`	37.08	25.9	4048
`bjork.mpg`	6.86	33.7	3824

Table 5.4: `mpeg_play` *with no display and full color dithering.*

Finally, we look at the time used to display the movie. `mpeg_play` uses shared memory images when possible, giving high throughput. The following command was executed, using `ordered2` dither[5]:

```
mpeg_play -framerate 0 -dither ordered2 -quiet file.mpg
```

Table 5.5 gives the results. Comparing this table to table 5.3, shows that the time used for displaying the movie is quite small, about 4% of the total time.

File	Time (sec)	FPS	RAM (kb)
`enterprise.mpg`	9.51	42.1	5008
`bart-temple.mpg`	34.60	27.7	4992
`bjork.mpg`	6.09	37.9	4848

Table 5.5: `mpeg_play` *with display and* `ordered2` *dither.*

mpeg2play-1.1b

As for `mpeg_play`, the first test was on decoding only, avoiding dithering and image output. The following command was used:

```
mpeg2play -o1 -q file.mpg
```

Table 5.6 shows the times used for pure MPEG frame decoding.

File	Time (sec)	FPS	RAM (kb)
`enterprise.mpg`	7.35	54.4	2656
`bart-temple.mpg`	26.88	35.7	2688
`bjork.mpg`	4.99	46.3	2560

Table 5.6: `mpeg2play` *with no display and no dithering.*

The program `mpeg2play` has no options for doing dithering without displaying the results, so we have to test dithering and displaying at the same time. The dithering done in `mpeg2play` matches the `ordered2` dither in `mpeg_play`. The following command was used:

```
mpeg2play -q file.mpg
```

Table 5.7 shows that dithering and displaying the images of the movie, takes 25–30% of the total execution time.

File	Time (sec)	FPS	RAM (kb)
`enterprise.mpg`	10.50	38.1	4112
`bart-temple.mpg`	35.42	27.1	4176
`bjork.mpg`	6.69	34.5	4080

Table 5.7: `mpeg2play` *with display and dithering.*

5.2.4 Results

Table 5.8 is a short summary of features provided by the two packages.

[5]The program was modified to quit at the end of the movie

	mpeg_play-2.3	mpeg2play-1.1b
Number of files	33	15
Lines of code	19 910	7 355
Bitplanes supported	8, 24, 32	8
Number of dithering algorithms	18	1
Avg. FPS, no dither	52.8	46.47

Table 5.8: Summary of features for the two MPEG-players.

The code of `mpeg_play` is more than twice as big as that of `mpeg2play`, which might fit with the fact that `mpeg_play` provides more dithering options, and support for more bitplanes. The higher speed of `mpeg_play`, combined with the support for more than eight bitplanes, makes it the packet of choice for further work.

5.3 Tailoring `mpeg_play`

A decision was made to use `mpeg_play-2.3-patched` as the origin for an MPEG playing code library. As an initial attempt to simplify the code, several of the available dithers were removed. Some of them didn't seem to work, while others failed to provide excessive features compared to the other dither algorithms. Several `#ifdef`'s that were not used, or that were used for testing, were removed, along with program options and features that were of no or little interest when using this program inside a Web browser, like the stand-alone control bar. The cleanup reduced the code size by several thousand lines.

Then came the time to extend the library. First of all, `mpeg_play` was written to read data from a file at it's own pace, that is; "pulling" data from the source. The input routines had to be rewritten to let Navigator "push" the data as it gets available. Secondly, `mpeg_play` spends some time decoding the data, especially if the flowing of the MPEG stream is slow. Since Navigator executes plug-ins in it's own thread, this could cause unfortunate "hangs", where user input is not responded to. This problem calls for a separate thread, and with separate threads come the problems of parallelism.

5.3.1 The Client — Server Approach

One of the two threads handles decoding of an MPEG stream to produce video output. This thread may be classified as an MPEG decoding *server*. The original thread, running inside Netscape Navigator, may be viewed as a *client*, feeding commands and MPEG data to the server.

Real multi-threading, that is; several processes running in the same address space, is currently not supported by all Unix-systems, so from a portability point of view, it is safer to rely on old-fashioned separate processes. This also gives another benefit: The `mpeg_play` source code pays little attention to freeing memory. For each movie it allocates lots of memory, but the chunks are never freed. When using a separate process for each film, the memory will automatically be released when the process terminates. Solving the problem by using separate processes is much easier than tracking every point of memory leakage in the code.

In Unix, the system call `fork` is used for creating a new process. The new (child) process is an exact copy of the calling (parent) process. Both processes continue running after the call to `fork`. The return value from `fork` makes it possible to identify which process is the parent, and which is the child, to let the two processes perform different tasks. The plug-in will `fork` off a child to handle the MPEG decoding, while the parent, running in Navigator's address space, will handle user input.

Interprocess Communication

The separate processes are supposed to cooperate, so we need a way to make them talk to each other through interprocess communication (IPC). The MPEG decoding process plays the role of a server, waiting for commands and data from the client process. The commands are initiated by user action, such as pressing buttons to start and stop the playback. Commands are also initiated by Navigator; for instance to quit the server when the user leaves a page, and when more MPEG data are available. All this traffic moves from the client to the server. We also need some information to go in the other direction: When Navigator works in a streamed manner, it will query the plug-in how many bytes it will accept (see `NPP_WriteReady` in section 5.1). The reception and buffering of incoming MPEG data is handled by the server, so we need to return that information through IPC.

Unix provides several methods for doing IPC (For an overview of all these methods and more, see [1] and [75].)

Shared files are regular files on disk. One process writes to the file, while the other reads. Obviously, this is quite slow, since it includes disk access.

Pipes are one way channels maintained by the operating system kernel. The processes use pipes via file handles obtained from the `pipe` system call. Pipes are read and written as regular files, but no disk access is involved. This kind of IPC is supported by all Unix systems, and it is the only one required by POSIX.1, an ISO standard operating system interface and environment to support application portability at the source code level.

Named pipes works like pipes, except that the entry point is a filename in the filesystem. This way processes that do not have the same ancestor may share the pipe.

Stream pipes are full duplex versions of pipes.

System V IPC has, as the name implies, it's origin on Unix System V, but are now available on most modern Unixes. Includes shared memory, semaphores, and message queues. Shared memory is the fastest way to transfer data between processes, since no copying is necessary. Unfortunately System V IPC resources are scarce; a limited number of shared memory segments, semaphores and message queues can exist at any given time on the system.

Ideally one would have used shared memory to transfer MPEG data, but since System V IPC may well not be available, the decision was made not to do so. Needless to say, shared files would be too slow, so the pipes were chosen. Since data is supposed to flow in both directions, two pipes are needed. Stream pipes were also an option, but it would give little benefit over "normal" pipes.

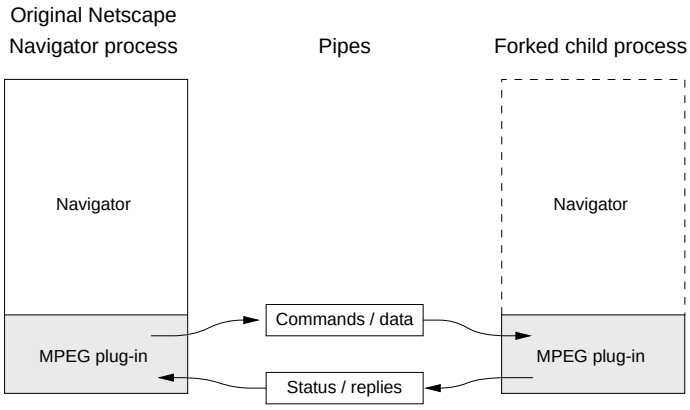

Figure 5.1: The two processes involved in the MPEG plug-in.

Figure 5.1 illustrates the two processes cooperating in decoding and viewing an MPEG stream. The process on the left is the original Netscape Navigator process with the plug-in dynamically loaded. When the plug-in starts receiving a stream, it forks, creating an identical process. The Navigator part of the new process is never run, and in modern operating systems it doesn't even take up valuable memory. Two pipes are created, one for sending commands and data from the client to the server, and one for giving feedback from the server to the client. When the client receives parts of the MPEG stream from Netscape Navigator, it passes it on to the server, and receives a status reply in return. The server handles all decoding, and displays the result in a subwindow of the browser.

5.3.2 The `mpeg_play` Library API

This section describes the API that was written for the extended `mpeg_play` library to hide the details of parallelism and IPC, and thus simplify it's use. The user of the library, in our case the MPEG plug-in, calls normal functions, of which two `fork` a new process, while others perform the IPC. The two `fork`'ing functions are the main entrypoints to the library. They differ in the way they set up the handling of the MPEG stream.

`mpNewStreamFromFile` takes a filename parameter, and is thus used when a local file containing the MPEG data is available.

`mpNewStreamPushed` initiates a pushed stream. The data transfer is controlled by the three functions `mpQueryWantedBytes`, `mpSendBytes` and `mpEndOfStream`, as described below.

After `fork`'ing, the child initializes the MPEG decoder. This includes setting up dithering, and creating a window in which to display the movie. It then enters the main loop of the program. This loop is responsible for fetching commands from the client, and do timing according to the pace of which the film is playing.

Among the commands accepted, are the ones used to push the MPEG stream to the decoder.

`mpQueryWantedBytes` is used to request the maximum number of bytes the library is able to receive at the time the call is issued.

`mpSendBytes` does the actual data transfer. Attempting to send more bytes than indicated by `mpQueryWantedBytes`, will result in an error.

`mpEndOfStream` tells the library that no more data will be sent for this stream.

Another set of functions is involved with direct user action. In the plug-in, these are called whenever the user presses buttons like "Play", "Rewind", "Stop", etc.

`mpSetLoop` sets or clears the internal loop flag. If the loop flag is set, the movie is automatically rewinded and restarted when the end is reached. Note that this is only possible if the MPEG stream is available to the library as a file.

`mpPlay` starts playing, or resumes playing from the current position.

`mpNextFrame` displays the next frame of the movie. Note that this is supposed to be used when the film is paused. When in play mode, the frames are advanced automatically.

`mpRewind` restarts the current stream. This is only possible when the library has access to the stream as a file.

`mpStop` will pause the playing.

`mpQuit` shuts down the server process. After this is called, no other functions should be used, since the child process is no longer available.

The last functions deal with the window in which the movie is shown. They are typically called when the client receives certain events from the window system.

`mpRepaint` should be called when it's time to redraw the contents of the window, typically when the parent window receives an `ExposureEvent`.

`mpParentResize` handles positioning of the video window within the parent. Should be called when the size of the parent is changed.

5.3.3 Avoiding the Pitfalls of Parallel Processing

The original `mpeg_play` was written to read the MPEG stream from a file. To simplify conversion to a more "event-driven" approach, a new set of functions were written, resembling the original `fread`- and `fgetc`-functions that were used formerly. The new functions work against a buffer that is filled when the client sends MPEG data using `mpSendBytes`. If the buffer is empty when any of these functions are called, a tight loop similar to the main loop is entered, waiting for more data from the client. The call to the reading function is not returned until more data is available, or an `mpEndOfStream` is issued.

The main goal of the tight loop, is to receive MPEG data for further decoding. What if the client sends other commands? Handling them may work for some, but fail for others. An example: The server is running out of data while decoding a frame, so it enters the loop to wait for more. The client sends a command ordering the server to skip to the next frame. The server obeys this command, calling the function to decode a frame *a second time* recursively, messing everything up. To get around this problem, queuing of commands was introduced. Every command that is not related to

sending MPEG data or quitting, are entered in a queue. This queue is later processed in the main loop of the server, before reading any further commands from the client.

Note that `mpQueryWantedBytes` must be responded to immediately, that is; not through the queue. Failure to do so would introduce a deadlock (more on deadlocks in [76]): The server is waiting for the client to send data. The client calls `mpQueryWantedBytes`, and waits for a reply. The reply will never come, since the server just queues the command.

5.3.4 On X11 and Colors

The X Window System[6] is designed to be portable across a wide range of platforms, with an even wider range of supported displays. Unfortunately, displays differ too much in their characteristics to make this transparent to programmers.

The display hardware offers one or more *bitplanes*. The combination of corresponding bits from each plane yields a pixel value, controlling a single pixel on the screen by indexing into a *colormap*. The number of simultaneous colors or grayscales, is thus given by the formula 2^n, where n is the number of bitplanes. Monochrome displays have a single bitplane. Color or grayscale displays typically have between 8 and 24 bitplanes.

Display hardware is typically capable of generating a much larger number of colors than may be displayed at once. To control which colors are currently displayable, *colormaps* are used. For color displays, a colormap entry describes the mixture of red, green and blue light used to produce the color in question. The pixel value from the bitplanes is used as an index into the current colormap.

Depending on the hardware, a colormap may be writable, or read-only. Writable colormaps let programs change the red, green and blue component to fit their needs. Read-only colormaps have preset values that may not be changed.

In X11, the characteristics of a colormap is described using a *visual*. The visual describes, among other things, the number of bitplanes, the size of the colormap, and a *visual class*. The visual class describes the features of the colormap; is it writable, or read-only? Is it color, grayscale or monochrome? Is the index into the colormap decomposed into separate indexes for the three color components? Table 5.9 sums this up for the six available visual classes:

Colormap type	Read/Write	Read-only
Monochrome/gray	`GrayScale`	`StaticGray`
Single index for R G & B	`PseudoColor`	`StaticColor`
Decomposed index for R G & B	`DirectColor`	`TrueColor`

Table 5.9: Comparison of Visual Classes. (From [77, section 7.3.4])

The books [77] and [78] gives thorough information on X11 and colors.

[6] http://www.x.org/

MPEG Plug-in and Colormaps

The MPEG plug-in has two approaches to the use of colors. Which one to use is decided by the user. If the hardware supports multiple colormaps, the plug-in may create it's own map, coexisting with the one used by Navigator. The switching of colormaps is done by the window manager, so applications using multiple colormaps have to inform the window manager about this. The information is passed using window manager hints on the toplevel window of the application. When the plug-in is set up to use its own map, it searches its ancestors until the main window is found, and adds the appropriate hints to that window.

A problem may arise when only one hardware colormap is available. In that case the plug-in has to share the colormap with Navigator. For displays with more than eight bitplanes, the numbers of available colors suffice. When only eight bitplanes are available, which is the case for many X Window workstations, only a few colors are available to the plug-in, as Navigator allocates most to itself. In our implementation, the plug-in allocates as many colors as it can. When further allocation fails, it starts matching against the existing entries in the colormap, to find close colors. Unfortunately, this yields unacceptable results.

5.4 Discussion

To be fully utilizable for end users, the MPEG plug-in still needs some fixing, but it is quite acceptable as a demonstration version. Most important, it fails on some MPEG-2 streams. This is true only when the `mpeg_play` library is used as a plug-in, not when running in a stand-alone test program. The cause for this problem is not known. The plug-in also lacks features expected in a high end commercial MPEG player, most notably sound. On Unix platforms, implementing portable audio is harder than implementing video, as there is no widely agreed upon interface to sound hardware. Also, sound handling must be implemented from scratch, as the original `mpeg_play` program had no support for it.

Another feature worth implementing, is the option to save the MPEG stream to a local file while playing, to enable rewinding, looping and faster playback. At the moment we can get around this by ordering Navigator to download the file *before* handing it to the plug-in, but it is impossible to play it at the same time it is downloaded to a file.

When sharing Navigator's colormap, the coloring of the movie is quite unacceptable. This could be fixed by implementing a smarter color allocation algorithm, and by making the dithering functions aware that the colors are not exactly matched. A better approach is to make Netscape extend their API to let plug-ins cooperate with Navigator on color allocation.

The performance seems to be more depending on the network connection than on the power of the CPU. For local files, the Navigator plug-in is approximately as fast as the `mpeg_play` program that it originated from.

5.5 Summary

The plug-in API defined by Netscape, defines two sets of functions: One residing within the browser, to be called by the plug-in, and one inside the plug-in to be called by the browser. The functions are used for communication between the browser and the plug-in.

An already existing MPEG decoder was chosen to do the MPEG decoding part of the plug-in. The choice was between two freely available coders, and after some testing, the decision fell on `mpeg_play`. Unnecessary code was removed, and the input routines were rewritten to allow a separate process to feed the data through pipes, instead of reading it from a file. A small API was defined for the MPEG decoder, to simplify connecting it to the plug-in code.

Due to the way Netscape Navigator reserves colors from the X Window System, the colors in the plug-in subwindow were not very good when the display hardware supported only one colormap.

Chapter 6

Sending Camera Input to a Java Applet

The implementation of a Java applet and accompaniment C-programs for receiving video from a remote camera, is described in this chapter. Due to time constraints, the main focus of the experiment was on Java programming and setting up and maintaining necessary connections, rather than doing a clever compressed transfer of image data. For a snapshot of the applet in action, see figure 4.4 on page 37. The source code[1] of all programs is available from WWW, and in appendixes. Appendix B on page 71 lists the Java applet source code of `SHHVid.class`, appendix C on page 77 shows the video grabber (`vidgrabber`) source, while appendix D on page 103 gives the proxy (`vidproxy`) listing.

The first section describes the communication across the network between the involved programs. Following that is a description of the simple video data transfer method used. The next section gives a few implementation details, while the final part presents some closing thoughts.

6.1 Network Communication

Figure 6.1 illustrates the use of three different programs to circumvent the security restrictions put on Java applets by their browsers.

Since the applet is not allowed to connect directly to the host providing the camera (as described in section 4.8.2 on page 37), a *proxy* is needed on the Web server from which the applet was loaded. The proxy functions as a gateway, redirecting packets from the camera server to any applets having announced their interest in the video feed.

[1] http://www.ifi.uio.no/~ftp/publications/cand-scient-theses/SHuseby/src/

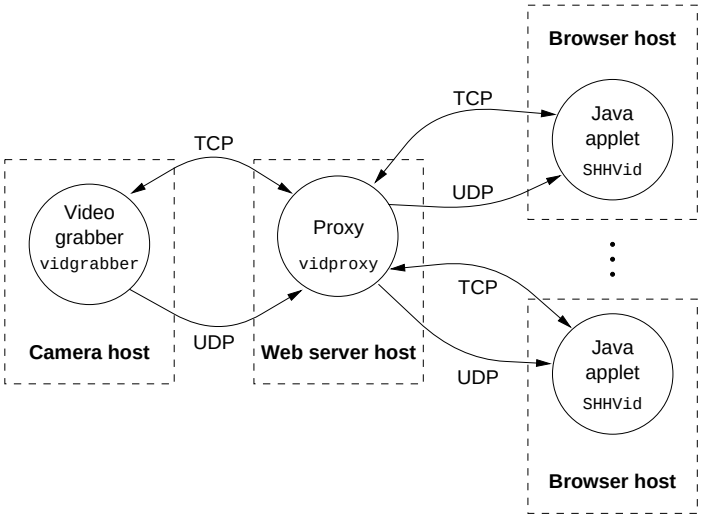

Figure 6.1: Three different programs running at three different hosts: The workaround for Java applet security.

The programs communicate using both TCP and UDP. The reliable, bidirectional TCP connections are used for passing control messages, like setting up and taking down the connection. The UDP "connection" is unidirectional. It is used for sending blocks of the image from the producer to the consumer. UDP was chosen for the image blocks to avoid the reliability overhead in TCP. Loosing a few packets is not critical, as the image is continuously updated.

The proxy running on the Web server is the program first started. It waits for incoming TCP connections from a video stream provider, and one or more viewers. When the grabber program is started on the host equipped with the camera, it connects to the proxy using TCP. If the proxy already has a camera feed, it responds with an error code, and shuts down the connection. Otherwise, it binds a local UDP port, and sends the port number to the grabber. Both programs wait, doing nothing until a viewer Java applet connects to the proxy.

When a viewer connects to the proxy using TCP, it may either receive an error code, or a welcome code. In the latter case, it binds a local UDP port, and sends the port number to the proxy. If the applet is the first to connect, the proxy tells the grabber to start the feed by sending a control code using TCP. The grabber sends datagrams containing image blocks to the proxy using the UDP port number received at startup, and the proxy resends all incoming datagrams to all viewers, using their respective UDP port numbers.

6.2 Video Handling

As was mentioned in the chapter introduction, the quality and size of the video transfer was not given priority. No standardized compression scheme was implemented. The following steps were taken to somewhat reduce the bandwidth requirements:

Quantization from eight to four bit grayscale, giving only 16 levels of gray. This way two pixels fit within a single byte. As no dithering is implemented to

compensate for this quantization, the visual quality of the video is rather low.

Spatial resolution reduction from 640×480, as sent from the video camera, to 160×120. The size is barely acceptable when used for viewing a closeup of a talking head.

Partitioning into blocks of 8×8 pixels, only sending blocks identified as changed. A very simple change detection scheme is implemented: A block is said to have changed if the maximum absolute single pixel difference compared to the previous block sent for the same image location exceeds some threshold value. This scheme is far from perfect, but it is easily implementable.

Ten times a second, the grabber program gets a frame from an IndyCam connected to a Silicon Graphics Indy, aided by the Video Library [79] shipped with the computer. As described above, the frame is analyzed for changed blocks, and chosen blocks are sent across the network, packed in UDP datagrams of at most 1400 bytes. Since some blocks may never or seldom change, every block that has not been sent in 5 seconds are resent, even if no change is measured. This is to make sure every viewer shows the entire picture.

The bandwidth requirements depends on the dynamics of the scene grabbed, and the sensibility of the camera. The worst case occurs when all blocks need to be resent ten times a second: A block occupies 35 bytes, including position within the image, and a block identifier. An image contains 20×15 blocks, giving a total of 10 500 bytes for a single frame. Ten frames per second thus requires a 0.8 Mbps line, not including overhead introduced by network packet encapsulation. According to figure 3.4 on page 21, neither analog modems nor ISDN, the two most likely connection types for home users, can cope with this. No negotiation of data transfer rate is implemented, the sender simply assumes that the network connection is capable of delivering the packets at the rate they are sent. Trying to send packets across a slow link, may fill the operating system's send queue, and temporarily stop the sending application.

6.3 Java Applet Implementation

On some browsers, a Java applet runs in the same thread as the browser itself. A CPU intensive applet will thus slow down the browser, giving delayed responses to user input in other parts of the program. The solution to this problem, is to let the main part of the applet run in a separate thread. The separate thread scenario is common enough that a special interface, called `Runnable`, is defined. Classes implementing this interface, must define a `run`-method that is called automatically when the thread is started.

To be able to support various kinds of image handling, Java defines the two interfaces `ImageProducer` and `ImageConsumer`. Classes implementing `ImageProducer` may generate images from different kinds of sources, and send them to classes implementing `ImageConsumer` for handling. The applet described in this chapter builds the image in an internal memory buffer as blocks are received from the network. The Java class `MemoryImageSource`, implementing an `ImageProducer`, is used to encapsulate the buffer in an `ImageProducer`, to be able to create a Java `Image` from it using the `createImage`-function. The `Image` representation is needed when ordering Java to

draw the image in a window. Note that `createImage` *copies* the buffer, so a new image must be created each time the internal buffer is changed.

Java supports various image color models by using subclasses of `ColorModel`. In our case, where only 16 levels of gray are needed, and indexed color model serves our needs. Java defines the class `IndexColorModel`, giving up to 256 simultaneous colors by using one byte pixel values for indexing. When initiating an instance of `IndexColorModel`, one has to specify the red, green and blue color component of each indexed color, along with an alpha value providing transparency. Our applet initiates the first 16 pixel values to represents shades of gray from black (pixel value 0) to white (pixel value 15). Once the `IndexColorModel` is set up and tied to the `MemoryImageSource`, the Java library silently handles all color mapping details behind the scene.

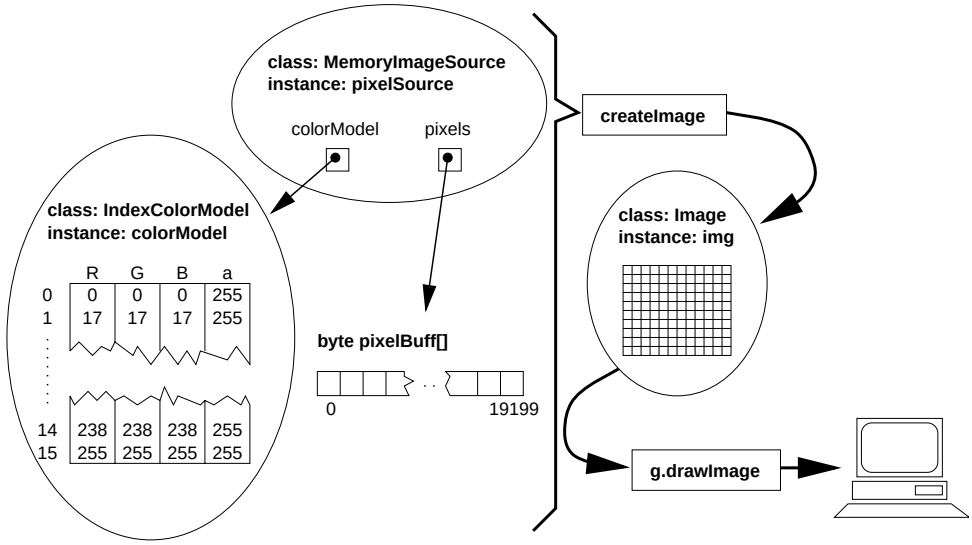

Figure 6.2: Conceptual representation of the interworking of image related classes in the video applet.

Figure 6.2 shows the connections between image handling classes in the applet. Instance names refers to the variable names used in the implementation. The `pixelBuff` array is the internal pixel buffer, in which updates to the image is made. Calls to `createImage` combines the values in the pixel buffer with the color values in the color model lookup table, creating a fully colored (or rather grayscaled for our purpose) representation of the image. Passing the image to the `drawImage` method of the `Graphics` context, results in the image being drawn on the screen.

Networking is done using BSD sockets [1] encapsulated in classes in the package (class collection) `java.net`. Every method working on sockets throw exceptions on error, so networking code must be embedded in `try`–`catch`-blocks to compile cleanly, even if handling some of the errors is not critical.

In this implementation, reading from and writing to a TCP connection is handled by the Java classes `InputStream` and `OutputStream` respectively, both instantiated inside the `Socket` class.

A separate Java class, `DatagramSocket`, is used for sockets referencing UDP "connections". This class provides the method `receive`, which is used to read a datagram into an instance of the Java class `DatagramPacket`. When creating instances of `DatagramPacket` classes, one has to provide a byte array to be used as a buffer, the size of which sets the maximum packet size to receive. When a package is received from the network, JDK shrinks the size of the buffer to match the packet read. This shrinking makes it impossible to reuse the `DatagramPacket` — a new one must be created with the correct buffer size.

6.4 Discussion

For some reason, the UDP-code in the applet causes a `SecurityException` when run within Netscape Navigator on a Silicon Graphics Indy equipped with the Irix operating system. No problems are encountered when using Navigator on Solaris, or Navigator and Microsoft Internet Explorer on Windows 95 and Windows NT. A bug in the Java library on Irix may be the cause to this problem. People at USIT reported similar behavior with their own Java UDP-code on HP-UX.

The Java applet had no problems receiving and decoding the image stream at the current rate and size. The inner loop in the decoding process is quite simple, doing three additions, three bitwise operations, two type casts, and two array element assignments. An interesting project for further study, would be to implement more advanced image representation schemes, and do performance tests on new implementations. If better compression is added, the size or number of packets transferred would be reduced, requiring less time to be spent in the network related code. It would be interesting to compare the time released by reducing the data size, to the time required to do more advanced decoding.

Moving this implementation from the experimental to the useful state, requires several enhancements. First of all, a standardized video compression scheme should be implemented, to allow higher visual quality and lower bandwidth requirements. Secondly, to be able to do more than wave and smile to the spectators, sound should be possibly transferred along with the video. Unfortunately, Java's sound capabilities are rather limited. Currently, the only sound support is the ability to play AU sound files, suitable for low quality, prerecorded clips.

Another important step is to take into account a possible limitation or variation in available bandwidth, by allowing negotiation of transfer speed. This greatly complicates the handling of viewer clients in the proxy and the grabber, since clients can no longer be handled equally. To cope with variations in bandwidth between viewer clients, some clients should get a reduced frame rate or a reduced quality compared to the others. In the current setup, the proxy just distributes datagrams to all viewers, with no knowledge of the contents. The grabber program handles all video details, but knows nothing about the various viewer clients. The easiest approach will probably be to extend the grabber to handle all viewer clients, and create adapted datagrams. This will however, greatly increase the network traffic between the grabber and the proxy.

6.5 Summary

Due to security constraints on Java applets, the network setup for delivery of camera output to an applet contains three programs on three different hosts: The video grabber running on the host with the camera sends it's data to a proxy running on the host providing the Web server. The proxy in turn hands the video data to the Java applet on the browser host.

Since no "real" compression was implemented, the size of the data was reduced by limiting the size of the image frames, and the number of colors (grayscales). In addition, care was taken not to send entire frames, but only parts that had changed since the previous frame.

Displaying images was left to instances of predefined classes in the Java library. A class providing an indexed color model removed the need to convert a single grayscale value to RGB values for each pixel.

Chapter 7

Conclusion

There are two groups of internationally accepted standards for video representation, the ITU-T recommendations H.261 and H.263, and the MPEG series. Both groups are represented on the Internet today: The ITU-T standards are implemented in video conferencing software along with "moving JPEG" and proprietary formats, while the two first MPEG standards, MPEG-1 and MPEG-2, typically are used for storing prerecorded video sequences. Freely available software encoders and decoders are available for both H.261 and MPEG, making it unnecessary to start entirely from scratch when implementing Web based services featuring these standards.

General data transfer methods may be used for video on demand systems when real-time play during transfer is not critical, that is, when the entire movie is downloaded to the local system before being played. General methods may also be used when the available bandwidth is sufficient to play the stream at real-time without modifying the quality of the movie.

Methods specialized in negotiating transfer options to cope with varying bandwidth exists, but most of them are either proprietary, or not through the entire standardization process. When using the Internet for video conferences, one also needs a set of protocols not directly related to video; protocols for session management.

Several mechanisms are available for extending Web browsers to support initially unsupported media types in general, and video in particular. The more promising methods seem to be plug-ins and Java applets, both supporting full integration within the browser window, and both being available in the more widely used browsers. A plug-in and a Java applet for video were implemented. The plug-in is capable of playing MPEG movies embedded in Netscape Navigator, while the Java applet receives and displays live video from a remote camera. The plug-in and the applet show that making video available from Web browsers is indeed possible, and not considerably harder than making a stand-alone video handling program.

A natural starting point for further work, is to extend the two programs implemented to make them fully usable. One of the main features lacking from both programs, is the ability to play audio along with the video. When combining video and audio, one has to take care of proper synchronization. For MPEG, synchronization issues are addressed in a separate part of the standard. For H.261/H.263, who are concerned with video only, audio and synchronization are specified in separate recommendations.

For full integration of various kinds of video inside Web browsers, some work will have to be left to the browser developers. The current plug-in model is built on the thought that what is transferred comes from a file on the remote end. The browser takes care of the transfer protocol details, and hands a stream of bytes to the plug-in that has announced it's capability of displaying the file type in question. Live video is typically not streamed, and real-time play requires continuous negotiation between the sender and the receiver, requiring other protocols than those used for regular file transfer. An extended plug-in model should not only allow plug-ins to announce what file types they can handle, but also any built in protocol support. Instead of sending a stream of bytes to such a plug-in, the browser would pass the URL, and leave it to the plug-in to parse it and take care of the communication.

Some extensions to the current Java API could simplify implementing video receivers. Writing decoders in Java is possible, but it is unclear how fast a pure Java implementation will be for the relatively advanced algorithms needed for H.261 and MPEG. Standardized class libraries for H.261 and MPEG would be very helpful, and could make access to hardware coders/decoders transparent to the user. Another useful approach would be to define classes for the CPU intensive tasks, such as the DCT transforms, motion compensation and statistical coding, and build coders and decoders using these tools. In addition, being able to grab video using Java without any user installed native code libraries, requires definition of classes for handling video camera input.

Appendix A

Introduction to Data Compression

This appendix is a short introduction to elementary and "well known" methods for compressing (coding) data. The goal of compression is to find a compact representation of information, by throwing away *redundancy*[1].

To have the necessary basement, we start by looking at some basic information theory. Following that is a description of statistical coding and dictionary based coding, two main categories for data compression methods.

This short survey is meant as a rough introduction only. More thorough information is found in the documents referenced in the text.

A.1 Basic Information Theory

An information source yields *symbols* from a finite *alphabet*. A sequence of symbols are commonly referenced as a *text*. An example of an information source is a dice that is thrown repeatedly. This source will generate a text from the alphabet $\{1, 2, 3, 4, 5, 6\}$.

To be able to measure information in one or more symbols from a source, we must know what is meant by *information*. Intuitively, a seldom occurring symbol will deliver more information than one seen often. A measure for the information found by the occurrence of a given symbol, must be the inverse of the probability of the occurrence.

Shannon defined the information from the occurrence of symbol s_i as

$$I(s_i) = \log \frac{1}{p(s_i)} = -\log p(s_i)$$

where $p(s_i)$ is the probability that the symbol s_i occurs.

In [80], the logarithm is explained by a wish that multiplied probability gives summed information, and a supporting example is given.

[1] **redundancy**: From latin. Exceeding what is necessary or normal.

Going one step further, we may talk about average information in a text, or *entropy*. Entropy is defined as

$$H = \sum_{i=1}^{n} p(s_i) I(s_i)$$

If we use a base two logarithm, the entropy is a theoretical lower limit for the average number of bits per symbol that is required to encode the stream of symbols.

In the above formulas, each occurrence of a symbol is taken as a statistically independent event. In reality, the probability of the occurrence of one symbol often depends on the occurrence of another. These so called Markov sources, require extended formulas for entropy, [12]. Markov models and general data compression are treated thoroughly in [81].

A.2 Compression Algorithms

The following sections are meant to give a quick introduction to basic compression algorithms. Algorithms for decompression are left out. For every algorithm, pointers to literature that further enlighten the topic are given.

A.2.1 Statistical Coding

A number of compression or coding algorithms assign codes of variable length to symbols, or groups of symbols depending on probability distribution. Examples of such algorithms include Huffman and Shannon-Fano coding. Implementations of these algorithms are called *statistical coders*. A non-uniform probability distribution is required for these coders to have any effect.

See [82] for complete explanation, and details of implementation of statistical coders. Also, the Frequently Asked Questions (FAQ) [83] from the comp.compression newsgroup may be helpful.

Statistical coders build code tables that are used for mapping between symbols and codes. Coders are often categorized by how this table is built[2].

Static coders use the same code table for everything that is to be coded. The table is generated once and for all, typically from representative samples of the data the coder is to be used on. The drawback of this approach is that all data differ from the representative data in varying degree, resulting in a non-optimal compression. The advantage is that no table needs to be bundled with each compressed stream. An example of a static code, is Morse code.

Semi-adaptive coders read the entire input stream to build the code table. Coding is done afterwards, requiring a second pass through the stream.

Dynamic or adaptive coders go one step further compared to the semi-adaptive coders. Code tables are built simultaneously with compression (and decompression). The advantages are that one doesn't need to transfer a separate code table, that the compression is adapted to local changes in the stream, and that

[2]Unfortunately, there is no "standard" for naming these categories.

the input stream is read only once. The drawbacks are that dynamic versions of the algorithms tend to be complicated, and that coders often are slower than the non-dynamic versions.

Huffman Coding

In Huffman coding, codes are generated with the aid of a binary tree according to the following algorithm:

1. Create a leaf node for every symbol, and let every node contain the probability of the occurrence of the symbol. The list of nodes is sorted on decreasing probability.
2. Create a new node based on the two orphan nodes with lowest probability, and make it the parent of the two nodes. The content of the new node is the sum of probabilities for the previously orphan nodes.
3. Repeat step 2 until only one orphan node exists. This node is the root of the tree.
4. Assign digits 0 and 1 to every left and right (or upper and lower, depending on the orientation of the tree) edge respectively.
5. To find the code of a symbol, follow each edge from the root node to the leaf node of the symbol, combining the digits on edges passed.

An important property of this method, is that codes are *instantaneously decodable*. This implies that the code for one symbol never occurs as a prefix of another, i.e. as soon as the decoder recognizes a code, it can convert it to a symbol.

Example We wish to code the text *"abracadabra"*. To code this without compression, we need three bits for each symbol, since the alphabet contains 5 symbols[3]. The uncompressed text will thus occupy $11 \times 3 = 33$ bits.

Let's see how much we can hope to achieve by calculating the entropy of the text. To do this, we first calculate the information of each symbol, and sum this up in table A.1.

Symbol s_i	Count	Prob $p(s_i)$	$I(s_i)$
a	5	0.455	1.138
b	2	0.182	2.459
r	2	0.182	2.459
d	1	0.091	3.459
c	1	0.091	3.459

Table A.1: Statistics for the text "abracadabra"

According to the formula in section A.1, the entropy thus becomes 2.040. Following the algorithm above, we build the tree in figure A.1

[3]Two bits would let us code four symbols, which doesn't suffice. Three bits let us code up to eight symbols, giving three unused code values. The codes we never use, show up as redundancy in our example

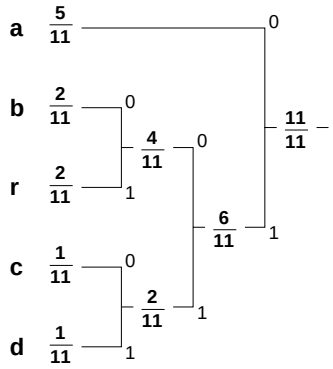

Figure A.1: An example Huffman tree for coding "abracadabra".

The algorithm states that one is to combine nodes containing the lowest probability. When several nodes with equal probability exists, one will have multiple choices. In figure A.1 the combinations is done as high in the tree as possible, to reduce the variance in the code length.

By traversing the tree, we end up with the codes in table A.2:

Symbol	Code
a	0
b	100
r	101
d	110
c	111

Table A.2: Huffman-codes for "abracadabra"

The text is coded to the sequence `01001010111011001001010`, containing 23 bits. On average, we have used 2.091 bits per symbol, quite close to the entropy. Compare this to the three bits used when coding directly without compression.

Arithmetic Coding

The drawback with Huffman coding, is that we assign an integer number of bits as the code for each symbol. The code is thus optimal when each symbol has an occurrence probability of 2^{-x} [12].

Arithmetic coding takes another approach. Instead of assigning a code for each symbol in the stream, the entire stream is given a single code. The code is a number in the interval $[\,0, 1\,)$, possibly containing a large number of digits. The coding is done by, for each symbol, shrinking the interval according to the following algorithm[4]:

1. Let the current interval be $[\,0, 1\,)$.
2. Go to step 5 if there are no more symbols in the input stream.

[4]Note that this is a simplification compared to a practical implementation

3. Split the current interval in subintervals, one for each symbol in the alphabet. Each subinterval should have a size reflecting the probability of occurrence of the symbol it represents.

4. Fetch the next symbol from the input stream, and shrink the current interval to the subinterval for the symbol read, then go to step 2.

5. Chose some number within the current interval to represent the entire stream.

Example Again, we wish to code the text *"abracadabra"*. The initial interval must thus be divided in the subintervals given in table A.3:

Symbol	Count	Lower	Upper
a	5	0	$\frac{5}{11}$
b	2	$\frac{5}{11}$	$\frac{7}{11}$
r	2	$\frac{7}{11}$	$\frac{9}{11}$
d	1	$\frac{9}{11}$	$\frac{9}{11}$
c	1	$\frac{10}{11}$	$\frac{11}{11}$

Table A.3: Subintervals for arithmetic coding of "abracadabra"

When coding, the first symbol shrinks the current interval to $[\,0, \frac{5}{11}\,)$. This interval is split in five new subintervals, one for each symbol in the alphabet. Further coding shrinks the interval more, finally giving us $[\,0.279383914419962, 0.279384089666912\,)$. The final code is some number within this interval. Optimally, we choose the number that can be coded with less binary digits. For our example this is the number 0.27938402, yielding the sequence `01000111100001011011011`, 23 bits. This gives no gain compared to the Huffman coding example. The reason for this is that the text is relatively short, and that the probability distribution is close to optimal for Huffman coding.

A.2.2 Dictionary Based Coding

Most variations on these dynamic coding algorithms, stem from one of two methods known as LZ77 and LZ78, which were published by Jacob Ziv and Abraham Lempel in *IEEE Transactions on Information Theory* [84] [85].

LZ77

Variations on LZ77 use previously seen text as a dictionary. The main structure in the original LZ77 is a two-part *sliding window*. The larger part of the window is text that is already coded, while the smaller part, called the *look-ahead buffer*, contains text that is to be coded. Incoming text is coded by tuples of the form *(index, length, successor symbol)*. *Index* points to a location within the window on which a match with some of the to-be-encoded text is found. The number of matching symbols is given by *length*, and *successor symbol* is the first symbol in the look-ahead buffer that doesn't match.

The algorithm resembles the following:

1. Fill the look-ahead buffer with symbols from the input stream.
2. Find the longest match between the (start of the) look-ahead buffer and the already seen text.
3. Output a tuple as described above to the output stream.
4. Shift the contents of the window *length + 1* symbols to the left, and fill empty bytes in the look-ahead buffer from the input stream.
5. Go to step 2 if the look-ahead buffer is not empty.

Example Figure A.2 shows the contents of the window during the last phase of the coding.

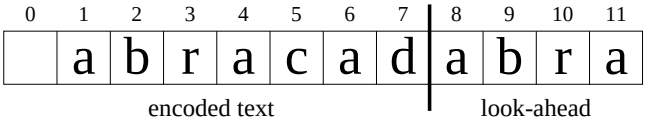

Figure A.2: The contents of the window at the final step of LZ77-coding of the text "abracadabra".

The first seven symbols are already coded and sent to the output stream, while the last four characters remain to be coded. By inspection, we see that the entire look-ahead buffer match text that is already seen. The final step of the coding thus yields the tuple *(1, 4, EOF)*, where *EOF* is a special symbol indicating the end of the stream. Since the window is 12 bytes long, we need four bits to encode an index (leaving four possible indexes unused). The look-ahead buffer is four bytes long, so we need two bits to represent the length if we allow zero lengths to be encoded using one of the unused index codes. The input alphabet contains five symbols plus the special *EOF* symbol, so three bits will suffice for this. Adding it all up, each tuple occupies nine bits.

When coding, the first three symbols each yield one tuple, while the next two pairs each give one tuple since *a* is found in the window. The four last symbols yield one tuple, giving a total of six tuples, and 54 bits.

Consult [82] for a number of different LZ77 variants, and what to do to increase the performance.

LZ78

Coders based on LZ78 takes another approach. Instead of using a window of previously seen text, they dynamically build a dictionary of previously seen *phrases*. New phrases are created by taking a previously stored phrase, and extending it with a single symbol. Initially, there is only one phrase with zero length, the nil-phrase. Like was the case with LZ77, output from an LZ78 coder consists of tuples. The tuples have the form *(phrase index, successor symbol)*. Coding is performed according to the following recipe:

1. Initiate the dictionary with a nil-phrase.

2. Let the nil-phrase be the current phrase.
3. Go to step 8 if there are no more symbols in the input stream.
4. Let the current symbol be the next symbol from the input stream.
5. If an existing phrase matches the current phrase extended with the current symbol, let the current phrase refer to this one in the dictionary, and go to step 3.
6. Send the tuple *(index to current phrase, current symbol)* to the output stream.
7. Create a new phrase in the dictionary containing the current phrase extended with the current symbol, and go to step 2.
8. Send the tuple *(index to current phrase, EOF)* to the output stream.

Example Once again, we code the text *"abracadabra"*. Table A.4 illustrates both output from, and dictionary build-up in the coder:

In	Out	New phrases	
		Index	Phrase
		0	""
a	*(0, a)*	1	"*a*"
b	*(0, b)*	2	"*b*"
r	*(0, r)*	3	"*r*"
a			
c	*(1, k)*	4	"*ac*"
a			
d	*(1, d)*	5	"*ad*"
a			
b	*(1, b)*	6	"*ab*"
r			
a	*(3, a)*	7	"*ra*"
	(0, EOF)		

Table A.4: Output from and dictionary generation for LZ78.

In this setup we need three bits to code the eight dictionary indexes, and three bits for the five symbols and the EOF special symbol. Each tuple thus gives six bits, making a total output of 48 bits.

Index encoding is done differently in the various LZ78 coders. It is possible to let the number of bits output from each tuple vary with the length of the dictionary [82].

Appendix B

SHHVid Java Applet Source Code

What follows is the source code to the Java video applet described in chapter 6.

```
1   import java.util.*;
2   import java.awt.*;
3   import java.awt.image.*;
4   import java.applet.*;
5   import java.io.*;
6   import java.net.*;
7
8   public class SHHVid extends Applet implements Runnable {
9       final int BLOCK_WIDTH = 8, BLOCK_HEIGHT = 8;
10      final int IMAGE_WIDTH = 160, IMAGE_HEIGHT = 120;
11      final int GRAY_BITS = 4, NUM_GRAY = 1 << GRAY_BITS;
12
13  /// error messages ////////////////////////////////////////////////
14      String errMessage = null;
15
16      private void errSetMessage(String s)
17      {
18          errMessage = s;
19          if (s != null)
20              System.err.println(s);
21          repaint();
22      }
23
24  /// image methods /////////////////////////////////////////////////
25      byte pixelBuff[] = new byte[IMAGE_WIDTH * IMAGE_HEIGHT];
26      MemoryImageSource pixelSource;
27      Image img;
28      IndexColorModel colorModel;
29
30      private void imgInit()
31      {
32          byte r[] = new byte[NUM_GRAY];
33          byte g[] = new byte[NUM_GRAY];
34          byte b[] = new byte[NUM_GRAY];
35          int q;
36
37          for (q = 0; q < NUM_GRAY; q++)
38              r[q] = g[q] = b[q] = (byte) ((q * 255) / (NUM_GRAY - 1));
```

```
39          colorModel = new IndexColorModel(8, NUM_GRAY, r, g, b);
40          for (q = 0; q < IMAGE_WIDTH * IMAGE_HEIGHT; q++)
41              pixelBuff[q] = 0;
42          pixelSource
43              = new MemoryImageSource(IMAGE_WIDTH, IMAGE_HEIGHT, colorModel,
44                                      pixelBuff, 0, IMAGE_WIDTH);
45          img = createImage(pixelSource);
46      }
47
48      private int imgSetBlock(int x, int y, byte buff[], int idx)
49      {
50          int xx, yy, imgIdx, lum;
51
52          imgIdx = (y * BLOCK_HEIGHT) * IMAGE_WIDTH + (x * BLOCK_WIDTH);
53          for (yy = BLOCK_HEIGHT; yy > 0; yy--) {
54              for (xx = BLOCK_WIDTH / 2; xx > 0; xx--) {
55                  pixelBuff[imgIdx++] = (byte) ((buff[idx] >> 4) & 0x0F);
56                  pixelBuff[imgIdx++] = (byte) (buff[idx++] & 0x0F);
57              }
58              imgIdx += (IMAGE_WIDTH - BLOCK_WIDTH);
59          }
60          return (BLOCK_WIDTH * BLOCK_HEIGHT) / 2;
61      }
62
63  /// networking methods ////////////////////////////////////////////
64      String remoteHost;
65      int remoteTcpPort = 8193;
66      Socket tcpSock = null;
67      DatagramSocket udpSock = null;
68      final byte PACK_NO_CAM = 0, PACK_BUSY = 1, PACK_SEND_PORT = 2;
69      final byte PACK_WELCOME = 3, PACK_GO_AWAY = 4;
70      final byte PACK_PORT = 10, PACK_HANGUP = 11;
71      final byte PACK_BLOCK = 20;
72      byte tcpPacketBuff[] = new byte[1400];
73      byte udpPacketBuff[] = new byte[1400];
74  //    long packetsReceived = 0;
75      DatagramPacket udpPacket;
76
77      private void netCloseConnection(boolean doHangup)
78      {
79          if (tcpSock == null)
80              return;
81          if (doHangup) {
82              System.err.println("sending hangup");
83              try {
84                  tcpPacketBuff[0] = PACK_HANGUP;
85                  tcpSock.getOutputStream().write(tcpPacketBuff, 0, 1);
86              } catch (IOException e) {
87                  errSetMessage("error sending data to server");
88              }
89          }
90          try {
91              tcpSock.close();
92              tcpSock = null;
93              if (udpSock != null) {
94                  udpSock.close();
95                  udpSock = null;
96              }
97          } catch (IOException e) {
98              System.err.println("error closing socket");
```

```
99          }
100     }
101
102     private boolean netOpenConnection()
103     {
104         byte packType = PACK_BUSY;
105         int port;
106
107         // get name of remote host
108         if ((remoteHost = getDocumentBase().getHost()).equals(""))
109             remoteHost = "www";
110         //    remoteHost = "localhost";
111         try {
112             errSetMessage("connecting to " + remoteHost
113                           + ":" + remoteTcpPort);
114             tcpSock = new Socket(remoteHost, remoteTcpPort);
115         } catch (UnknownHostException e) {
116             errSetMessage("unknown host: " + remoteHost);
117             return false;
118         } catch (IOException e) {
119             errSetMessage("unable to connect to " + remoteHost);
120             return false;
121         }
122         try {
123             tcpSock.getInputStream().read(tcpPacketBuff);
124             packType = tcpPacketBuff[0];
125         } catch (IOException e) {
126             errSetMessage("unable to read from server");
127             netCloseConnection(false);
128             return false;
129         }
130         if (packType == PACK_NO_CAM) {
131             errSetMessage("no camera available");
132             netCloseConnection(false);
133             return false;
134         } else if (packType == PACK_BUSY) {
135             errSetMessage("server is busy");
136             netCloseConnection(false);
137             return false;
138         } else if (packType == PACK_SEND_PORT) {
139             try {
140                 udpSock = new DatagramSocket();
141                 port = udpSock.getLocalPort();
142                 System.out.println("local udp port is " + port);
143                 tcpPacketBuff[0] = PACK_PORT;
144                 tcpPacketBuff[1] = (byte) ((port >> 8) & 0xFF);
145                 tcpPacketBuff[2] = (byte) (port & 0xFF);
146                 tcpSock.getOutputStream().write(tcpPacketBuff, 0, 3);
147                 tcpSock.getInputStream().read(tcpPacketBuff);
148                 packType = tcpPacketBuff[0];
149                 if (packType == PACK_WELCOME) {
150                     System.out.println("we're connected");
151                     errSetMessage(null);
152                 } else if (packType == PACK_GO_AWAY) {
153                     errSetMessage("server doesn't want us.");
154                     netCloseConnection(false);
155                     return false;
156                 }
157             } catch (SocketException e) {
158                 errSetMessage("unable to set up local udp socket");
```

```
159                 netCloseConnection(false);
160                 return false;
161             } catch (IOException e) {
162                 errSetMessage("error sending data to server");
163                 netCloseConnection(false);
164                 return false;
165             }
166         } else {
167             errSetMessage("got unknown packet -- closing");
168             netCloseConnection(false);
169             return false;
170         }
171         return true;
172     }
173 
174     private void netUdpInput()
175     {
176         int  n, idx, x, y;
177         byte type;
178 
179         try {
180             // the length of the buffer was shrinked by Java
181             // to match each packet received, so it must be reset for
182             // each read operation.
183             udpPacket = new DatagramPacket(udpPacketBuff,
184                                            udpPacketBuff.length);
185             udpSock.receive(udpPacket);
186         } catch (IOException e) {
187             if (runner != null) {
188                 errSetMessage("error receiving packet");
189                 netCloseConnection(true);
190             }
191             return;
192         }
193 //      if (++packetsReceived % 50 == 0)
194 //          System.out.println("received " + packetsReceived + " packets");
195         n = udpPacket.getLength();
196         idx = 0;
197         while (idx < n) {
198             type = udpPacketBuff[idx++];
199             if (type == PACK_BLOCK) {
200                 x = udpPacketBuff[idx++];
201                 y = udpPacketBuff[idx++];
202                 idx += imgSetBlock(x, y, udpPacketBuff, idx);
203             } else {
204                 errSetMessage("unknown packet received");
205                 netCloseConnection(true);
206                 return;
207             }
208         }
209         // for some reason, the createImage function must be called
210         // whenever a change is done. i assume there is another
211         // way to do this.
212         img = createImage(pixelSource);
213         repaint();
214     }
215 
216 /// "ordinary" applet methods ////////////////////////////////////////
217     Thread runner = null;
218 
```

```
219     public String getAppletInfo() {
220         return "SHHVid by Sverre H. Huseby";
221     }
222 
223     private int getIntParam(String name, int dflt) {
224         String param = getParameter(name);
225         if (param == null) {
226 System.out.println("port set to default");
227             return dflt;
228         } else
229 System.out.println("port set to " + param);
230         return Integer.parseInt(param);
231     }
232 
233     public void init()
234     {
235         remoteTcpPort = getIntParam("port", remoteTcpPort);
236         imgInit();
237     }
238 
239     public void start() {
240         errSetMessage(null);
241         netOpenConnection();
242         if (runner == null) {
243             runner = new Thread(this);
244             runner.start();
245         }
246     }
247 
248     public void stop() {
249         if (runner != null)
250             runner.stop();
251         runner = null;
252         netCloseConnection(true);
253     }
254 
255     public void run()
256     {
257         while(runner != null && tcpSock != null) {
258             netUdpInput();
259 //          Thread.yield();
260         }
261     }
262 
263     public void paint(Graphics g)
264     {
265         g.drawImage(img, 0, 0, this);
266         if (errMessage != null) {
267             g.setColor(Color.red);
268             g.drawString(errMessage, 10, 14);
269         }
270     }
271 
272     public void update(Graphics g)
273     {
274         paint(g);
275     }
276 
277     public boolean mouseDown(Event e, int x, int y)
278     {
```

```
279         if (runner == null)
280             start();
281         else
282             stop();
283         return true;
284     }
285 }
286
```

Appendix C

SHHVid Grabber Source Code

This appendix contains the source code to the camera grabber used in conjunction with the Java video applet described in chapter 6. Note that the program uses general purpose libraries developed by the author of this report. The source code to these libraries are not included. All source code may be found at
`http://www.ifi.uio.no/~ftp/publications/cand-scient-theses/SHuseby/src/`

The following files are listed.

- `Makefile`
- `vidgrabber.h` and `vidgrabber.c`
- `camera.h` and `camera.c`
- `blockimg.h` and `blockimg.c`
- `network.h` and `network.c`
- `shhsched.h` and `shhsched.c`

Makefile

```
1   # $Id$
2   PROG            = vidgrabber
3   DIST            = $(PROG)
4   VERMAJ          = 0
5   VERMIN          = 1
6   VERPAT          = 0
7   VERSION         = $(VERMAJ).$(VERMIN).$(VERPAT)
8   COMPILED_DATE   = `date '+%Y-%m-%d %H:%M:%S'`
9   COMPILED_BY     = `whoami`
10
11  ###############################################################################
12
13  # where are shhmsg and shhopt?
14  INCDIR          = -I/hom/sverrehu/c/net/shhnet \
15                    -I/usr/local/include -I$$HOME/include
16
17  LIBDIR          = -L/hom/sverrehu/c/net/shhnet \
18                    -L/usr/local/lib -L$$HOME/lib/$$HOSTTYPE
19
20
```

```
21  INSTBASEDIR     = /usr/local
22  INSTBINDIR      = $(INSTBASEDIR)/bin
23  INSTMANDIR      = $(INSTBASEDIR)/man/man1
24  INSTALL         = install -m 644
25  INSTALLPROG     = install -s -m 755
26  MKDIRP          = install -d -m 755
27
28  DEFINES         = -DVERSION=\"$(VERSION)\" \
29                    "-DCOMPILED_DATE=\"$(COMPILED_DATE)\"" \
30                    "-DCOMPILED_BY=\"$(COMPILED_BY)\""
31
32  ###############################################################################
33
34  CC              = gcc
35
36  OPTIM           = -O2
37  CCOPT           = -s -Wall $(OPTIM) $(INCDIR) $(DEFINES) $(CFLAGS)
38  LDOPT           = -s $(LIBDIR) $(LDFLAGS)
39
40  LIBS            = -lshhmsg -lshhopt -lshhnet -lvl
41  OBJS            = blockimg.o camera.o network.o shhsched.o vidgrabber.o
42
43  ###############################################################################
44
45  all: $(PROG)
46
47  $(PROG): $(OBJS)
48
49  .o: $(OBJS)
50          $(CC) $(CCOPT) -o $@ $(OBJS) $(LDOPT) $(LIBS)
51
52  .c.o:
53          $(CC) -o $@ -c $(CCOPT) $<
54
55  clean:
56          rm -f *.o core depend *~
57
58  install: $(PROG)
59          $(MKDIRP) $(INSTBINDIR) $(INSTMANDIR)
60          $(INSTALLPROG) $(PROG) $(INSTBINDIR)
61          $(INSTALL) $(PROG).1 $(INSTMANDIR)
62
63  depend dep:
64          $(CC) $(INCDIR) -MM *.c >depend
65
66  ###############################################################################
67
68  # To let the author make a distribution. The rest of the Makefile
69  # should be used by the author only.
70  DISTDIR         = $(DIST)-$(VERSION)
71  DISTFILE        = $(DIST)-$(VERSION).tar.gz
72  DISTFILES       = Makefile \
73                    blockimg.c camera.c network.c shhsched.c vidgrabber.c \
74                    blockimg.h camera.h network.h shhsched.h vidgrabber.h
75
76  chmod:
77          chmod -R a+rX *
78
79  veryclean: clean
80          rm -f $(PROG) $(DIST)-$(VERSION).tar.gz gmon.out
```

```
81
82  dist: chmod
83          mkdir $(DISTDIR)
84          chmod a+rx $(DISTDIR)
85          for q in $(DISTFILES); do \
86            if test -r $$q; then \
87              ln -s ../$$q $(DISTDIR); \
88            else echo "warning: no file $$q"; fi; \
89            done
90          tar -cvhzf $(DISTFILE) $(DISTDIR)
91          chmod a+r $(DISTFILE)
92          rm -rf $(DISTDIR)
93
94  ifeq (depend,$(wildcard depend))
95  include depend
96  endif
```

vidgrabber.h

```
1   /* $Id$ */
2   #ifndef VIDGRABBER_H
3   #define VIDGRABBER_H
4
5   /* milliseconds between each */
6   #define SEND_DELAY    100
7
8   void vidStartGrab(void);
9   void vidStopGrab(void);
10
11  #endif
```

vidgrabber.c

```
1   /* $Id$ */
2   /************************************************************************
3    *
4    *  FILE            vidgrabber.c
5    *  MODULE OF       vidgrabber
6    *
7    *  DESCRIPTION
8    *
9    *  WRITTEN BY      Sverre H. Huseby <sverrehu@ifi.uio.no>
10   *
11   ************************************************************************/
12
13  #include <stdlib.h>
14  #include <stdio.h>
15
16  #include <shhmsg.h>
17  #include <shhopt.h>
18  #include <shhnet.h>
19
20  #include "camera.h"
21  #include "shhsched.h"
22  #include "network.h"
23  #include "blockimg.h"
24  #include "vidgrabber.h"
25
26  /************************************************************************
27   *                                                                      *
28   *                     P R I V A T E   D A T A                          *
29   *                                                                      *
30   ************************************************************************/
31
32  static BlockImage blkImgOut;
33  static SchedId    grabSchedId;
34
35
36
37  /************************************************************************
38   *                                                                      *
39   *                  P R I V A T E   F U N C T I O N S                   *
40   *                                                                      *
41   ************************************************************************/
42
43  static void
```

```
44  vidGrabAndSend(void *clientData)
45  {
46      grabSchedId = schedAddTime(SEND_DELAY, vidGrabAndSend, NULL);
47      camGetFrame(&blkImgOut);
48      netSendImage(&blkImgOut);
49  }
50
51  static void
52  version(void)
53  {
54      printf(
55        "%s " VERSION ", by Sverre H. Huseby "
56        "(compiled " COMPILED_DATE " by " COMPILED_BY ")\n",
57        msgGetName()
58      );
59      exit(0);
60  }
61
62  static void
63  usage(void)
64  {
65      printf(
66        "usage: %s [options] remote-host[:remote-port]\n"
67        "\n"
68        "  -h, --help      display this help and exit\n"
69        "  -V, --version   output version information and exit\n"
70        "\n",
71        msgGetName()
72      );
73      exit(0);
74  }
75
76
77
78  /*****************************************************************************
79   *                                                                           *
80   *                    P U B L I C    F U N C T I O N S                       *
81   *                                                                           *
82   *****************************************************************************/
83
84  void
85  vidStartGrab(void)
86  {
87      grabSchedId = schedAddTime(SEND_DELAY, vidGrabAndSend, NULL);
88  }
89
90  void
91  vidStopGrab(void)
92  {
93      schedRemoveTime(grabSchedId);
94  }
95
96  int
97  main(int argc, char *argv[])
98  {
99      optStruct opt[] = {
100       /* short long           type        var/func         special      */
101         { 'h', "help",        OPT_FLAG,   usage,           OPT_CALLFUNC },
102         { 'V', "version",     OPT_FLAG,   version,         OPT_CALLFUNC },
103         { 0, 0, OPT_END, 0, 0 }  /* no more options */
```

```
104     };
105
106     msgSetName(argv[0]);
107     snSetErrorHandling(SN_REPORT_ERROR_AND_EXIT, msgFatal);
108
109     optParseOptions(&argc, argv, opt, 0);
110     if (argc != 2)
111         usage();
112
113     netSetRemoteSystem(argv[1]);
114
115     schedInit();
116     camInit();
117     netInit();
118
119     schedLoop();
120
121     netFinish();
122     camFinish();
123     schedFinish();
124     return 0;
125 }
```

camera.h

```
1   /* $Id$ */
2   #ifndef CAMERA_H
3   #define CAMERA_H
4
5   #include "blockimg.h"
6
7   void camInit(void);
8   void camFinish(void);
9
10  void camGetFrame(BlockImage *blockImage);
11
12  #endif
```

camera.c

```
1   /* $Id$ */
2   /*************************************************************************
3    *
4    *  FILE            camera.c
5    *  MODULE OF       vidgrabber
6    *
7    *  DESCRIPTION
8    *
9    *  WRITTEN BY      Sverre H. Huseby <sverrehu@ifi.uio.no>
10   *
11   *************************************************************************/
12
13  #include <stdio.h>
14  #include <string.h>
15  #if defined(sgi)
16  #include <vl/vl.h>
17  #endif
18
19  #include <shhmsg.h>
20
21  #include "blockimg.h"
22  #include "camera.h"
23
24  #undef SAVEFILE "save.img"
25  #if !defined(sgi)
26  #define LOADFILE "capture.img"
27  #endif
28
29  /*************************************************************************
30   *                                                                       *
31   *                     P R I V A T E   D A T A                           *
32   *                                                                       *
33   *************************************************************************/
34
35  #if defined(LOADFILE)
36  static FILE *camFile;
37  static int camWidth = IMAGE_WIDTH;
38  #elif defined(sgi)
39  static char *camDisplayName = "";
40  static VLServer camServer;
41  static VLNode camSource, camDrain;
42  static VLPath camPath;
```

```
43  static VLBuffer camBuffer;
44  static int camWidth, camHeight;
45  #ifdef SAVEFILE
46  static FILE *camFile;
47  #endif
48  #endif
49
50
51
52  /*****************************************************************************
53   *                                                                           *
54   *                  P R I V A T E   F U N C T I O N S                        *
55   *                                                                           *
56   *****************************************************************************/
57
58  static void
59  camFatal(void)
60  {
61  #if defined(sgi)
62      msgFatal("camera error: %s\n", vlStrError(vlErrno));
63  #endif
64  }
65
66
67
68  /*****************************************************************************
69   *                                                                           *
70   *                   P U B L I C   F U N C T I O N S                         *
71   *                                                                           *
72   *****************************************************************************/
73
74  void
75  camInit(void)
76  {
77  #if defined(sgi)
78      VLControlValue cv;
79  #endif
80
81  #if defined(LOADFILE)
82      if ((camFile = fopen(LOADFILE, "rb")) == NULL)
83          msgFatalPerror(LOADFILE);
84  #elif defined(sgi)
85      if ((camServer = vlOpenVideo(camDisplayName)) == NULL)
86          camFatal();
87      if ((camSource = vlGetNode(camServer, VL_SRC, VL_VIDEO, VL_ANY)) < 0)
88          camFatal();
89      if ((camDrain = vlGetNode(camServer, VL_DRN, VL_MEM, VL_ANY)) < 0)
90          camFatal();
91      if ((camPath = vlCreatePath(camServer, VL_ANY, camSource, camDrain)) < 0)
92          camFatal();
93      if (vlSetupPaths(camServer, (VLPathList) &camPath,
94                       1, VL_LOCK, VL_LOCK) < 0)
95          camFatal();
96      cv.intVal = VL_PACKING_Y_8_P;
97      if (vlSetControl(camServer, camPath, camDrain, VL_PACKING, &cv) < 0)
98          camFatal();
99      cv.fractVal.numerator = 1;
100     cv.fractVal.denominator = 4;
101     if (vlSetControl(camServer, camPath, camDrain, VL_ZOOM, &cv) < 0)
102         camFatal();
```

```
103     cv.xyVal.x = IMAGE_WIDTH;
104     cv.xyVal.y = IMAGE_HEIGHT;
105     if (vlSetControl(camServer, camPath, camDrain, VL_SIZE, &cv) < 0)
106         camFatal();
107     if ((camBuffer = vlCreateBuffer(camServer, camPath, camDrain, 1)) == NULL)
108         camFatal();
109     if (vlRegisterBuffer(camServer, camPath, camDrain, camBuffer) < 0)
110         camFatal();
111     vlGetControl(camServer, camPath, camDrain, VL_SIZE, &cv);
112     camWidth = cv.xyVal.x;
113     camHeight = cv.xyVal.y;
114 #ifdef SAVEFILE
115     if ((camFile = fopen(SAVEFILE, "wb")) == NULL)
116         msgFatalPerror(SAVEFILE);
117 #endif
118 #endif
119 }
120
121 void
122 camFinish(void)
123 {
124 #if defined(LOADFILE)
125     fclose(camFile);
126 #elif defined(sgi)
127     vlDeregisterBuffer(camServer, camPath, camDrain, camBuffer);
128     vlDestroyBuffer(camServer, camBuffer);
129     vlDestroyPath(camServer, camPath);
130     vlCloseVideo(camServer);
131 #ifdef SAVEFILE
132     fclose(camFile);
133 #endif
134 #endif
135 }
136
137 void
138 camGetFrame(BlockImage *blockImage)
139 {
140 #if defined(LOADFILE) || defined(sgi)
141     int x, y, yb;
142     register int xb;
143     register unsigned char *d, *s;
144     unsigned char tmpbuff[BLOCK_WIDTH * BLOCK_HEIGHT];
145 #endif
146 #if defined(LOADFILE)
147     static unsigned char buffer[IMAGE_WIDTH * IMAGE_HEIGHT];
148 #elif defined(sgi)
149     unsigned char *buffer;
150     VLInfoPtr info;
151     VLTransferDescriptor xferDesc;
152 #endif
153
154 #if defined(LOADFILE)
155     if (fread(buffer, IMAGE_WIDTH * IMAGE_HEIGHT, 1, camFile) < 1) {
156         rewind(camFile);
157         fread(buffer, IMAGE_WIDTH * IMAGE_HEIGHT, 1, camFile);
158     }
159 #elif defined(sgi)
160     vlPutFree(camServer, camBuffer);
161     xferDesc.mode = VL_TRANSFER_MODE_DISCRETE;
162     xferDesc.count = 1;
```

```
163     xferDesc.delay = 0;
164     xferDesc.trigger = VLTriggerImmediate;
165     if (vlBeginTransfer(camServer, camPath, 1, &xferDesc) == -1)
166         camFatal();
167     while ((info = vlGetLatestValid(camServer, camBuffer)) == NULL) {
168         if (vlGetErrno())
169             camFatal();
170     }
171     vlEndTransfer(camServer, camPath);
172     buffer = vlGetActiveRegion(camServer, camBuffer, info);
173 #ifdef SAVEFILE
174     fwrite(buffer, IMAGE_WIDTH * IMAGE_HEIGHT, 1, camFile);
175 #endif
176 #endif
177 #if defined(LOADFILE) || defined(sgi)
178     for (y = 0; y < VER_BLOCKS; y++)
179         for (x = 0; x < HOR_BLOCKS; x++) {
180             d = tmpbuff;
181             for (yb = 0; yb < BLOCK_HEIGHT; yb++) {
182                 s = buffer + camWidth * (y * BLOCK_HEIGHT + yb)
183                     + x * BLOCK_WIDTH;
184                 for (xb = BLOCK_WIDTH; xb; xb--)
185 #if 0
186                     *d++ = (*s++ & 0xFF) >> (8 - GRAY_BITS);
187 #else
188                     *d++ = *s++;
189 #endif
190             }
191             blkCalcMeasure(&(blockImage->block[x][y]), tmpbuff);
192             memcpy(blockImage->block[x][y].data, tmpbuff, sizeof(tmpbuff));
193         }
194 #endif
195 }
```

blockimg.h

```
1   /* $Id$ */
2   #ifndef BLOCKIMG_H
3   #define BLOCKIMG_H
4
5   #define GRAY_BITS       4
6   #define NUM_GRAY        (1 << GRAY_BITS)
7   /* block size must be dividable by two */
8   #define BLOCK_WIDTH     8
9   #define BLOCK_HEIGHT    8
10
11  /* the image sizes must be multipla of block sizes */
12  #define IMAGE_WIDTH   160
13  #define IMAGE_HEIGHT  120
14
15  #define HOR_BLOCKS (IMAGE_WIDTH / BLOCK_WIDTH)
16  #define VER_BLOCKS (IMAGE_HEIGHT / BLOCK_HEIGHT)
17
18  #define MEASURE_MAX_DELTA  5
19  #define RESEND_FREQUENCY   50
20
21  typedef struct {
22      int measure;
23      int measureLastSend;
24      int lastImageNumber;
25      unsigned char data[BLOCK_WIDTH * BLOCK_HEIGHT];
26  } Block;
27
28  typedef struct {
29      Block block[HOR_BLOCKS][VER_BLOCKS];
30  } BlockImage;
31
32  void blkCalcMeasure(Block *block, unsigned char *data);
33
34  #endif
```

blockimg.c

```
1   /* $Id$ */
2   /*****************************************************************************
3    *
4    *  FILE            blockimg.c
5    *  MODULE OF
6    *
7    *  DESCRIPTION
8    *
9    *  WRITTEN BY      Sverre H. Huseby <sverrehu@ifi.uio.no>
10   *
11   *****************************************************************************/
12
13  #include <stdlib.h>
14
15  #include "blockimg.h"
16
17  /*****************************************************************************
18   *                                                                           *
19   *                    P U B L I C    F U N C T I O N S                       *
20   *                                                                           *
```

```
21   ************************************************************************/
22
23  void
24  blkCalcMeasure(Block *block, unsigned char *data)
25  {
26  #if 1
27      register unsigned char *p1, *p2;
28      register int           diff;
29      int                    q, maxdiff = 0;
30
31      p1 = block->data;
32      p2 = data;
33      for (q = BLOCK_WIDTH * BLOCK_HEIGHT; q; q--) {
34          diff = *p1++ - *p2++;
35          if (diff < 0)
36              diff = -diff;
37          if (diff > maxdiff)
38              maxdiff = diff;
39      }
40      block->measure = maxdiff;
41  #else
42      register unsigned char *p1, *p2;
43      register int           diff;
44      int                    q, totdiff = 0;
45
46      p1 = block->data;
47      p2 = data;
48      for (q = BLOCK_WIDTH * BLOCK_HEIGHT; q; q--) {
49          diff = *p1++ - *p2++;
50          if (diff < 0)
51              diff = -diff;
52          totdiff += diff;
53      }
54      block->measure = totdiff / (BLOCK_WIDTH * BLOCK_HEIGHT);
55  #endif
56  }
```

network.h

```
1   /* $Id$ */
2   #ifndef NETWORK_H
3   #define NETWORK_H
4
5   #include "blockimg.h"
6
7   #define DEFAULT_CAMERA_PORT 8194
8
9   #define MAX_PACKET 1400
10
11  /* packet types */
12  enum {
13      /* tcp server */
14      PACK_HAVE_FEED = 30,  /* already have video feed */
15      PACK_WELCOME_PORT,    /* call accepted, port number included */
16      PACK_START_GRAB,      /* start grabbing and sending frames */
17      PACK_STOP_GRAB,       /* stop grabbing and sending frames */
18      /* tcp client */
19      PACK_QUIT = 40,       /* end of call */
20      /* udp */
21      PACK_BLOCK = 20,      /* a block of an image */
22  };
23
24  void netSetRemoteSystem(const char *system);
25
26  void netInit(void);
27  void netFinish(void);
28
29  void netSendImage(BlockImage *blockImage);
30
31  #endif
```

network.c

```
1   /* $Id$ */
2   /*****************************************************************************
3    *
4    *  FILE            network.c
5    *  MODULE OF       vidgrabber
6    *
7    *  DESCRIPTION
8    *
9    *  WRITTEN BY      Sverre H. Huseby <sverrehu@ifi.uio.no>
10   *
11   *****************************************************************************/
12
13  #include <stdlib.h>
14  #include <stdio.h>
15  #include <string.h>
16  #include <unistd.h>
17
18  #include <shhmsg.h>
19  #include <shhnet.h>
20
21  #include "blockimg.h"
22  #include "camera.h"
23  #include "shhsched.h"
```

```
24  #include "vidgrabber.h"
25  #include "network.h"
26
27  /***************************************************************************
28   *                                                                         *
29   *                      P R I V A T E   D A T A                            *
30   *                                                                         *
31   ***************************************************************************/
32
33  /* need some extra space */
34  static unsigned char netOutPacket[MAX_PACKET + BLOCK_WIDTH * BLOCK_HEIGHT];
35  static int    netOutIndex;
36  static int    netOutImageNumber;
37
38  static char   netRemoteSystem[81] = "";
39  static char   netRemoteHost[81];
40  static char   netRemoteTcpPort[81];
41  static int    netRemoteTcp = -1;
42
43  static SNAddr netRemoteUdpAddr;
44  static SNPort netRemoteUdpPort;
45  static int    netRemoteUdp = -1;
46
47  SchedId netTcpReadyId;
48
49
50
51  /***************************************************************************
52   *                                                                         *
53   *                  P R I V A T E   F U N C T I O N S                      *
54   *                                                                         *
55   ***************************************************************************/
56
57  static void
58  netTcpInput(void *clientData, int fd)
59  {
60      unsigned char buff[128];
61      char          name[100];
62      int           n, type;
63      SNPort        dummy;
64
65      if (tcpRead(fd, buff, sizeof(buff), &n) != SN_OK || n == 0)
66          msgFatal("got error or empty packet, assuming broken connection\n");
67
68      type = buff[0];
69      switch (type) {
70        case PACK_HAVE_FEED:
71          msgFatal("another camera provider running\n");
72          break;
73        case PACK_WELCOME_PORT:
74          inGetPeerAddrPort(fd, &netRemoteUdpAddr, &dummy);
75          inHostAddrToName(netRemoteUdpAddr, name, sizeof(name));
76          netRemoteUdpPort = ((int) buff[1] << 8) | buff[2];
77          udpClientSock(&netRemoteUdp);
78          msgMessage("connected to %s, sending UDP data to port %u\n",
79                     name, (unsigned) netRemoteUdpPort);
80          break;
81        case PACK_START_GRAB:
82          msgMessage("starting grab on request\n");
83          vidStartGrab();
```

```
84            break;
85          case PACK_STOP_GRAB:
86            msgMessage("stopping grab on request\n");
87            vidStopGrab();
88            break;
89          default:
90            msgError("unknown packet received (%u)\n", (unsigned) type);
91      }
92  }
93
94  static void
95  netFlush(void)
96  {
97      int n;
98
99      if (!netOutIndex)
100         return;
101     if (netRemoteUdp >= 0) {
102         udpWriteTo(netRemoteUdp, netOutPacket, netOutIndex,
103                    netRemoteUdpAddr, netRemoteUdpPort, &n);
104         if (n < netOutIndex)
105             msgError("partial packet sent\n");
106     }
107     netOutIndex = 0;
108 }
109
110 static void
111 netSendBlock(Block *block, int x, int y)
112 {
113 #if GRAY_BITS == 4
114     unsigned char          pix;
115 #endif
116     int                    n, origOutIndex, nBytes;
117     unsigned char          buff[BLOCK_WIDTH * BLOCK_HEIGHT];
118     register unsigned char *s, *d;
119
120     s = block->data;
121     d = buff;
122 #if GRAY_BITS == 4
123     nBytes = (BLOCK_WIDTH * BLOCK_HEIGHT) / 2;
124     for (n = nBytes; n; n--) {
125         pix = *s++ & 0xF0;
126         *d++ = pix | (*s++ >> 4);
127     }
128 #else
129     nBytes = BLOCK_WIDTH * BLOCK_HEIGHT;
130     for (n = nBytes; n; n--)
131         *d++ = *s++ >> (8 - GRAY_BITS);
132 #endif
133
134     if (netOutIndex + 3 + BLOCK_WIDTH * BLOCK_HEIGHT > MAX_PACKET)
135         netFlush();
136     origOutIndex = netOutIndex;
137     for (;;) {
138         netOutPacket[netOutIndex++] = PACK_BLOCK;
139         netOutPacket[netOutIndex++] = (unsigned char) x;
140         netOutPacket[netOutIndex++] = (unsigned char) y;
141         memcpy(netOutPacket + netOutIndex, buff, nBytes);
142         netOutIndex += nBytes;
143         if (netOutIndex > MAX_PACKET) {
```

```
144             netOutIndex = origOutIndex;
145             netFlush();
146             origOutIndex = netOutIndex;
147         } else
148             break;
149     }
150     block->measureLastSend = block->measure;
151     block->lastImageNumber = netOutImageNumber;
152 }
153 
154 static void
155 netCallRemote(void)
156 {
157     msgMessage("calling proxy at %s:%s\n", netRemoteHost, netRemoteTcpPort);
158     tcpClientOpen(netRemoteHost, netRemoteTcpPort, &netRemoteTcp);
159     netTcpReadyId
160         = schedAddDesc(SCHED_READ_READY, netRemoteTcp, netTcpInput, NULL);
161 }
162 
163 static void
164 netGetHostAndPort(const char *system, char *host, char *port)
165 {
166     const char *p;
167 
168     if ((p = strchr(system, ':')) == NULL) {
169         strcpy(host, system);
170         sprintf(port, "%d", DEFAULT_CAMERA_PORT);
171     } else {
172         if (p == system)
173             strcpy(host, "localhost");
174         else
175             strncpy(host, system, p - system);
176         strcpy(port, p + 1);
177         if (!strlen(port))
178             sprintf(port, "%d", DEFAULT_CAMERA_PORT);
179     }
180 }
181 
182 
183 
184 /*************************************************************************
185  *                                                                       *
186  *                  P U B L I C   F U N C T I O N S                      *
187  *                                                                       *
188  *************************************************************************/
189 
190 void
191 netSetRemoteSystem(const char *system)
192 {
193     strcpy(netRemoteSystem, system);
194 }
195 
196 void
197 netInit(void)
198 {
199     netOutIndex = 0;
200     netOutImageNumber = RESEND_FREQUENCY;
201 
202     netGetHostAndPort(netRemoteSystem, netRemoteHost, netRemoteTcpPort);
203     netCallRemote();
```

```
204 }
205
206 void
207 netFinish(void)
208 {
209     if (netRemoteTcp >= 0) {
210         tcpClose(netRemoteTcp);
211         netRemoteTcp = -1;
212     }
213     if (netRemoteUdp >= 0) {
214         tcpClose(netRemoteUdp);
215         netRemoteUdp = -1;
216     }
217     schedRemoveDesc(netTcpReadyId);
218 }
219
220 void
221 netSendImage(BlockImage *blockImage)
222 {
223     int x, y, diff;
224     Block *block;
225
226     for (y = 0; y < VER_BLOCKS; y++)
227         for (x = 0; x < HOR_BLOCKS; x++) {
228             block = &(blockImage->block[x][y]);
229             if (netOutImageNumber - block->lastImageNumber
230                 >= RESEND_FREQUENCY) {
231                 netSendBlock(block, x, y);
232             } else {
233                 diff = block->measure - block->measureLastSend;
234                 if (diff < 0)
235                     diff = -diff;
236                 if (diff > MEASURE_MAX_DELTA)
237                     netSendBlock(block, x, y);
238             }
239         }
240     netFlush();
241     ++netOutImageNumber;
242 }
```

shhsched.h

```
1   /* $Id$ */
2   #ifndef SHHSCHED_H
3   #define SHHSCHED_H
4
5   #include <sys/time.h>  /* struct timeval */
6
7   #ifdef __cplusplus
8     extern "C" {
9   #endif
10
11  typedef unsigned long SchedId;
12
13  /*
14   *  function calls at given times
15   */
16  typedef void (*SchedTimeFunc)(void *clientData);
17  typedef unsigned long SchedTime; /* milliseconds */
18
19  SchedId schedAddTimeAt(struct timeval *when, SchedTimeFunc func,
20                         void *clientData);
21  SchedId schedAddTime(SchedTime delay, SchedTimeFunc func, void *clientData);
22  void    schedRemoveTime(SchedId id);
23  void    schedRemoveTimeAll(void);
24
25
26  /*
27   *  function calls when file descriptors are ready
28   */
29  typedef void (*SchedDescFunc)(void *clientData, int fd);
30  typedef enum {
31      SCHED_READ_READY,       /* file descriptor ready for reading */
32      SCHED_WRITE_READY,      /* file descriptor ready for writing */
33      SCHED_EXCEPTION_READY   /* exception waiting on file descriptor */
34  } SchedDescAction;
35
36  SchedId schedAddDesc(SchedDescAction action, int fd,
37                       SchedDescFunc func, void *clientData);
38  void    schedRemoveDesc(SchedId id);
39  void    schedRemoveDescAll(void);
40
41
42  /*
43   *  main loop, and setting up.
44   */
45  void    schedInit(void);
46  void    schedFinish(void);
47  void    schedEndLoop(void);
48  void    schedLoop(void);
49
50
51  #ifdef __cplusplus
52    }
53  #endif
54
55  #endif
```

shhsched.c

```
1   /* $Id$ */
2   /**************************************************************************
3    *
4    *  FILE            shhsched.c
5    *
6    *  DESCRIPTION
7    *
8    *  WRITTEN BY      Sverre H. Huseby <sverrehu@ifi.uio.no>
9    *
10   **************************************************************************/
11
12  #include <stdlib.h>
13  #include <stdio.h>
14  #include <errno.h>
15  #include <sys/time.h>
16  #include <sys/types.h>
17  #include <unistd.h>
18  #if defined(sgi)
19  #include <bstring.h>
20  #endif
21
22  #include <shhmsg.h>
23
24  #include "shhsched.h"
25
26  /* the following should be changed to a dynamic scheme later */
27  #define MAX_SCHED_TIME_FUNCS 20
28  #define MAX_SCHED_DESC_FUNCS FD_SETSIZE
29
30  /**************************************************************************
31   *                                                                        *
32   *                      P R I V A T E    D A T A                          *
33   *                                                                        *
34   **************************************************************************/
35
36  typedef struct _SchedTimeItem {
37      SchedId                id;
38      SchedTimeFunc          func;
39      void                  *clientData;
40      struct timeval         when;
41      struct _SchedTimeItem *next;
42  } SchedTimeItem;
43
44  typedef struct _SchedDescItem {
45      SchedId                id;
46      SchedDescFunc          func;
47      void                  *clientData;
48      int                    fd;
49      SchedDescAction        action;
50      struct _SchedDescItem *next;
51  } SchedDescItem;
52
53  static SchedTimeItem  schedTimePool[MAX_SCHED_TIME_FUNCS];
54  static SchedTimeItem *schedTimeFree;
55  static SchedTimeItem *schedTimeNext;
56
57  static SchedDescItem  schedDescReadPool[MAX_SCHED_DESC_FUNCS];
58  static SchedDescItem *schedDescReadFree;
```

```
59  static SchedDescItem *schedDescReadNext;
60  static SchedDescItem  schedDescWritePool[MAX_SCHED_DESC_FUNCS];
61  static SchedDescItem *schedDescWriteFree;
62  static SchedDescItem *schedDescWriteNext;
63  static SchedDescItem  schedDescExceptPool[MAX_SCHED_DESC_FUNCS];
64  static SchedDescItem *schedDescExceptFree;
65  static SchedDescItem *schedDescExceptNext;
66  static fd_set schedDescReadSet;
67  static fd_set schedDescWriteSet;
68  static fd_set schedDescExceptSet;
69
70  static SchedId schedLastId;
71
72  static int schedKeepLooping;
73
74
75
76  /*****************************************************************************
77   *                                                                           *
78   *                  P U B L I C   F U N C T I O N S                          *
79   *                                                                           *
80   *****************************************************************************/
81
82  SchedId
83  schedAddTimeAt(struct timeval *when, SchedTimeFunc func, void *clientData)
84  {
85      register SchedTimeItem *ti, *ti2;
86
87      if ((ti = schedTimeFree) == NULL)
88          msgFatal("too many functions scheduled for later calls\n");
89      schedTimeFree = ti->next;
90      ti->id = ++schedLastId;
91      ti->func = func;
92      ti->clientData = clientData;
93      ti->when = *when;
94      /* insert at correct location. assumes that tv_usec < 1 second */
95      if ((ti2 = schedTimeNext) == NULL
96          || ti2->when.tv_sec > ti->when.tv_sec
97          || (ti2->when.tv_sec == ti->when.tv_sec
98              && ti2->when.tv_usec > ti->when.tv_usec)) {
99          ti->next = schedTimeNext;
100         schedTimeNext = ti;
101     } else {
102         while (ti2->next) {
103             if (ti2->next->when.tv_sec > ti->when.tv_sec
104                 || (ti2->next->when.tv_sec == ti->when.tv_sec
105                     && ti2->next->when.tv_usec > ti->when.tv_usec))
106                 break;
107             ti2 = ti2->next;
108         }
109         ti->next = ti2->next;
110         ti2->next = ti;
111     }
112     return ti->id;
113 }
114
115 SchedId
116 schedAddTime(SchedTime delay, SchedTimeFunc func, void *clientData)
117 {
118     struct timeval when;
```

```
119
120     /* calculate the time to call this function */
121     gettimeofday(&when, NULL);
122     when.tv_sec += delay / 1000L;
123     when.tv_usec += ((delay % 1000L) * 1000L);
124     if (when.tv_usec >= 1000000L) {
125         when.tv_usec -= 1000000L;
126         ++when.tv_sec;
127     }
128     return schedAddTimeAt(&when, func, clientData);
129 }
130
131 void
132 schedRemoveTime(SchedId id)
133 {
134     SchedTimeItem *ti, **prev;
135
136     if ((ti = schedTimeNext) == NULL)
137         return;
138     prev = &schedTimeNext;
139     do {
140         if (ti->id == id) {
141             *prev = ti->next;
142             ti->next = schedTimeFree;
143             schedTimeFree = ti;
144             break;
145         }
146         prev = &ti->next;
147     } while ((ti = ti->next) != NULL);
148 }
149
150 void
151 schedRemoveTimeAll(void)
152 {
153     int q;
154
155     schedTimeNext = NULL;
156     schedTimeFree = &schedTimePool[0];
157     for (q = 0; q < MAX_SCHED_TIME_FUNCS - 1; q++)
158         schedTimePool[q].next = &schedTimePool[q + 1];
159     schedTimePool[q].next = NULL;
160 }
161
162 SchedId
163 schedAddDesc(SchedDescAction action, int fd,
164              SchedDescFunc func, void *clientData)
165 {
166     SchedDescItem *di = NULL, **nextp = NULL, **freep = NULL;
167     fd_set *fds = NULL;
168
169     switch (action) {
170       case SCHED_READ_READY:
171         fds = &schedDescReadSet;
172         freep = &schedDescReadFree;
173         nextp = &schedDescReadNext;
174         break;
175       case SCHED_WRITE_READY:
176         fds = &schedDescWriteSet;
177         freep = &schedDescWriteFree;
178         nextp = &schedDescWriteNext;
```

```
179         break;
180       case SCHED_EXCEPTION_READY:
181         fds = &schedDescExceptSet;
182         freep = &schedDescExceptFree;
183         nextp = &schedDescExceptNext;
184         break;
185       default:
186         msgFatal("illegal action in schedAddDesc\n");
187     }
188     if ((di = *freep) == NULL)
189         msgFatal("too many descriptor functions scheduled\n");
190     *freep = di->next;
191     di->id = ++schedLastId;
192     di->func = func;
193     di->clientData = clientData;
194     di->fd = fd;
195     di->action = action;
196     di->next = *nextp;
197     *nextp = di;
198     FD_SET(fd, fds);
199     return di->id;
200 }
201
202 void
203 schedRemoveDesc(SchedId id)
204 {
205     int q;
206     SchedDescItem *di = NULL, **prev = NULL, **freep = NULL;
207     fd_set *fds = NULL;
208
209     for (q = 0; q < 3; q++) {
210         switch (q) {
211           case 0:
212             di = schedDescReadNext;
213             prev = &schedDescReadNext;
214             freep = &schedDescReadFree;
215             fds = &schedDescReadSet;
216             break;
217           case 1:
218             di = schedDescWriteNext;
219             prev = &schedDescWriteNext;
220             freep = &schedDescWriteFree;
221             fds = &schedDescWriteSet;
222             break;
223           case 2:
224             di = schedDescExceptNext;
225             prev = &schedDescExceptNext;
226             freep = &schedDescExceptFree;
227             fds = &schedDescExceptSet;
228             break;
229         }
230         if (di == NULL)
231             continue;
232         do {
233             if (di->id == id) {
234                 FD_CLR(di->fd, fds);
235                 *prev = di->next;
236                 di->next = *freep;
237                 *freep = di;
238                 goto finish;
```

```
239             }
240             prev = &di->next;
241         } while ((di = di->next) != NULL);
242     }
243   finish:
244 }
245
246 void
247 schedRemoveDescAll(void)
248 {
249     int q;
250
251     schedDescReadNext = NULL;
252     schedDescReadFree = &schedDescReadPool[0];
253     for (q = 0; q < MAX_SCHED_DESC_FUNCS - 1; q++)
254         schedDescReadPool[q].next = &schedDescReadPool[q + 1];
255     schedDescReadPool[q].next = NULL;
256     FD_ZERO(&schedDescReadSet);
257
258     schedDescWriteNext = NULL;
259     schedDescWriteFree = &schedDescWritePool[0];
260     for (q = 0; q < MAX_SCHED_DESC_FUNCS - 1; q++)
261         schedDescWritePool[q].next = &schedDescWritePool[q + 1];
262     schedDescWritePool[q].next = NULL;
263     FD_ZERO(&schedDescWriteSet);
264
265     schedDescExceptNext = NULL;
266     schedDescExceptFree = &schedDescExceptPool[0];
267     for (q = 0; q < MAX_SCHED_DESC_FUNCS - 1; q++)
268         schedDescExceptPool[q].next = &schedDescExceptPool[q + 1];
269     schedDescExceptPool[q].next = NULL;
270     FD_ZERO(&schedDescExceptSet);
271 }
272
273 void
274 schedInit(void)
275 {
276     schedRemoveTimeAll();
277     schedRemoveDescAll();
278     schedLastId = 0;
279 }
280
281 void
282 schedFinish(void)
283 {
284 }
285
286 void
287 schedEndLoop(void)
288 {
289     schedKeepLooping = 0;
290 }
291
292 /*-------------------------------------------------------------------------
293  *
294  *  NAME          schedLoop
295  *
296  *  FUNCTION      The main loop of the scheduler.
297  *
298  *  SYNOPSIS      #include "shhsched.h"
```

```
299  *               void (void);
300  *
301  *  INPUT
302  *
303  *  OUTPUT
304  *
305  *  RETURNS
306  *
307  *  DESCRIPTION
308  */
309 void
310 schedLoop(void)
311 {
312     int ret;
313     /* the reason for using 0,1 instead of 0,0 for timeoutnow, is that
314      * SGI select didn't return > 0 for 0,0. */
315     struct timeval now, timeout, *tvp, timeoutnow = { 0, 0 };
316     fd_set fdsr, fdsw, fdse;
317     SchedTimeItem *ti;
318     SchedDescItem *di;
319
320     schedKeepLooping = 1;
321     while (schedKeepLooping) {
322         /* need a loop to restart the select if interrupted by a signal. */
323         do {
324             /* find next time function to call. */
325             if ((ti = schedTimeNext) == NULL)
326                 tvp = NULL;  /* no function registered, sleep 'forever' */
327             else {
328                 gettimeofday(&now, NULL);
329                 /* calculate timeout value */
330                 timeout.tv_sec = ti->when.tv_sec - now.tv_sec;
331                 if ((timeout.tv_usec = ti->when.tv_usec - now.tv_usec) < 0L) {
332                     timeout.tv_usec += 1000000L;
333                     --timeout.tv_sec;
334                 }
335                 /* check if already expired */
336                 if (timeout.tv_sec < 0 || timeout.tv_usec < 0)
337                     tvp = &timeoutnow;
338                 else
339                     tvp = &timeout;
340             }
341             fdsr = schedDescReadSet;
342             fdsw = schedDescWriteSet;
343             fdse = schedDescExceptSet;
344             ret = select(FD_SETSIZE, &fdsr, &fdsw, &fdse, tvp);
345             if (ret == 0) {
346                 /* timeout */
347                 /* call function */
348                 ti->func(ti->clientData);
349                 /* remove function from list */
350                 schedTimeNext = ti->next;
351                 ti->next = schedTimeFree;
352                 schedTimeFree = ti;
353             } else if (ret > 0) {
354                 /* some descriptor ready */
355                 /* first check the read set */
356                 di = schedDescReadNext;
357                 while (di) {
358                     if (FD_ISSET(di->fd, &fdsr)) {
```

```
359                         di->func(di->clientData, di->fd);
360                         if (--ret == 0)
361                             goto all_called;
362                     }
363                     di = di->next;
364                 }
365                 /* then check the write set */
366                 di = schedDescWriteNext;
367                 while (di) {
368                     if (FD_ISSET(di->fd, &fdsw)) {
369                         di->func(di->clientData, di->fd);
370                         if (--ret == 0)
371                             goto all_called;
372                     }
373                     di = di->next;
374                 }
375                 /* and finally check the exception set */
376                 di = schedDescExceptNext;
377                 while (di) {
378                     if (FD_ISSET(di->fd, &fdse)) {
379                         di->func(di->clientData, di->fd);
380                         if (--ret == 0)
381                             goto all_called;
382                     }
383                     di = di->next;
384                 }
385               all_called:
386             }
387         } while (ret < 0 && errno == EINTR);
388     }
389 }
```

Appendix D

SHHVid Proxy Source Code

This appendix contains the source code to the video proxy used in conjunction with the Java video applet described in chapter 6. Note that the program uses general purpose libraries developed by the author of this report. The source code to these libraries are not included. All source code may be found at
`http://www.ifi.uio.no/~ftp/publications/cand-scient-theses/SHuseby/src/`

The following files are listed.

- `Makefile`
- `network.h` and `vidproxy.c`
- `netcamera.h` and `netcamera.c`
- `netviewer.h` and `netviewer.c`
- `shhsched.h` and `shhsched.c`

Makefile

```
1   # $Id$
2   PROG            = vidproxy
3   DIST            = $(PROG)
4   VERMAJ          = 0
5   VERMIN          = 1
6   VERPAT          = 0
7   VERSION         = $(VERMAJ).$(VERMIN).$(VERPAT)
8   COMPILED_DATE   = `date '+%Y-%m-%d %H:%M:%S'`
9   COMPILED_BY     = `whoami`
10
11  ###############################################################################
12
13  # where are shhmsg and shhopt?
14  INCDIR          = -I/hom/sverrehu/c/net/shhnet \
15                    -I/usr/local/include -I$$HOME/include
16
17  LIBDIR          = -L/hom/sverrehu/c/net/shhnet \
18                    -L/usr/local/lib -L$$HOME/lib/$$HOSTTYPE
19
20
21  INSTBASEDIR     = /usr/local
22  INSTBINDIR      = $(INSTBASEDIR)/bin
```

```
23  INSTMANDIR      = $(INSTBASEDIR)/man/man1
24  INSTALL         = install -m 644
25  INSTALLPROG     = install -s -m 755
26  MKDIRP          = install -d -m 755
27
28  DEFINES         = -DVERSION=\"$(VERSION)\" \
29                    "-DCOMPILED_DATE=\"$(COMPILED_DATE)\"" \
30                    "-DCOMPILED_BY=\"$(COMPILED_BY)\""
31
32  ###############################################################################
33
34  CC              = gcc
35
36  OPTIM           = -O2
37  CCOPT           = -s -Wall $(OPTIM) $(INCDIR) $(DEFINES) $(CFLAGS)
38  LDOPT           = -s $(LIBDIR) $(LDFLAGS)
39
40  LIBS            = -lshhmsg -lshhopt -lshhnet
41  OBJS            = netcamera.o netviewer.o shhsched.o vidproxy.o
42
43  ###############################################################################
44
45  all: $(PROG)
46
47  $(PROG): $(OBJS)
48
49  .o: $(OBJS)
50          $(CC) $(CCOPT) -o $@ $(OBJS) $(LDOPT) $(LIBS)
51
52  .c.o:
53          $(CC) -o $@ -c $(CCOPT) $<
54
55  clean:
56          rm -f *.o core depend *~
57
58  install: $(PROG)
59          $(MKDIRP) $(INSTBINDIR) $(INSTMANDIR)
60          $(INSTALLPROG) $(PROG) $(INSTBINDIR)
61          $(INSTALL) $(PROG).1 $(INSTMANDIR)
62
63  depend dep:
64          $(CC) $(INCDIR) -MM *.c >depend
65
66  ###############################################################################
67
68  # To let the author make a distribution. The rest of the Makefile
69  # should be used by the author only.
70  DISTDIR         = $(DIST)-$(VERSION)
71  DISTFILE        = $(DIST)-$(VERSION).tar.gz
72  DISTFILES       = Makefile \
73                    netcamera.c netviewer.c shhsched.c vidproxy.c \
74                    netcamera.h netviewer.h network.h shhsched.h
75
76  chmod:
77          chmod -R a+rX *
78
79  veryclean: clean
80          rm -f $(PROG) $(DIST)-$(VERSION).tar.gz gmon.out
81
82  dist: chmod
```

```
83          mkdir $(DISTDIR)
84          chmod a+rx $(DISTDIR)
85          for q in $(DISTFILES); do \
86            if test -r $$q; then \
87              ln -s ../$$q $(DISTDIR); \
88            else echo "warning: no file $$q"; fi; \
89            done
90          tar -cvhzf $(DISTFILE) $(DISTDIR)
91          chmod a+r $(DISTFILE)
92          rm -rf $(DISTDIR)
93
94  ifeq (depend,$(wildcard depend))
95  include depend
96  endif
```

network.h

```
1   /* $Id$ */
2   #ifndef NETWORK_H
3   #define NETWORK_H
4
5   #define DEFAULT_VIEWER_PORT 8193
6   #define DEFAULT_CAMERA_PORT 8194
7
8   #define MAX_PACKET 1400
9
10  #endif
```

vidproxy.c

```
1   /* $Id$ */
2   /*****************************************************************************
3    *
4    *  FILE            vidproxy.c
5    *  MODULE OF       vidproxy
6    *
7    *  DESCRIPTION
8    *
9    *  WRITTEN BY      Sverre H. Huseby <sverrehu@ifi.uio.no>
10   *
11   *****************************************************************************/
12
13  #include <stdlib.h>
14  #include <stdio.h>
15
16  #include <shhmsg.h>
17  #include <shhopt.h>
18  #include <shhnet.h>
19
20  #include "shhsched.h"
21  #include "netcamera.h"
22  #include "netviewer.h"
23
24  /*****************************************************************************
25   *                                                                           *
26   *                      P R I V A T E   D A T A                              *
27   *                                                                           *
28   *****************************************************************************/
29
30
31
32  /*****************************************************************************
33   *                                                                           *
34   *                  P R I V A T E   F U N C T I O N S                        *
35   *                                                                           *
36   *****************************************************************************/
37
38  static void
39  version(void)
40  {
41      printf(
42        "%s " VERSION ", by Sverre H. Huseby "
43        "(compiled " COMPILED_DATE " by " COMPILED_BY ")\n",
44        msgGetName()
```

```
45      );
46      exit(0);
47  }
48
49  static void
50  usage(void)
51  {
52      printf(
53        "usage: %s [options]\n"
54        "\n"
55        "  -h, --help               display this help and exit\n"
56        "  -c, --camera-port=PORT   set the port number for camera daemon\n"
57        "  -p, --port=PORT          set the port number for remote viewers\n"
58        "  -V, --version            output version information and exit\n"
59        "\n",
60        msgGetName()
61      );
62      exit(0);
63  }
64
65
66
67  /**************************************************************************
68   *                                                                        *
69   *                  P U B L I C   F U N C T I O N S                       *
70   *                                                                        *
71   **************************************************************************/
72
73  int
74  main(int argc, char *argv[])
75  {
76      optStruct opt[] = {
77        /* short long           type        var/func          special       */
78          { 'h', "help",        OPT_FLAG,   usage,            OPT_CALLFUNC },
79          { 'c', "camera-port", OPT_STRING, netcSetLocalPort, OPT_CALLFUNC },
80          { 'p', "port",        OPT_STRING, netvSetLocalPort, OPT_CALLFUNC },
81          { 'V', "version",     OPT_FLAG,   version,          OPT_CALLFUNC },
82          { 0, 0, OPT_END, 0, 0 }  /* no more options */
83      };
84
85      msgSetName(argv[0]);
86      snSetErrorHandling(SN_REPORT_ERROR_AND_EXIT, msgFatal);
87
88      optParseOptions(&argc, argv, opt, 0);
89
90      schedInit();
91      netcInit();
92      netvInit();
93
94      schedLoop();
95
96      netvFinish();
97      netcFinish();
98      schedFinish();
99      return 0;
100 }
```

netcamera.h

```
1   /* $Id$ */
2   #ifndef NETCAMERA_H
3   #define NETCAMERA_H
4
5   #include "network.h"
6
7   /* packet types */
8   enum {
9       /* tcp server */
10      PACK_HAVE_FEED = 30,  /* already have video feed */
11      PACK_WELCOME_PORT,    /* call accepted, port number included */
12      PACK_START_GRAB,      /* start grabbing and sending frames */
13      PACK_STOP_GRAB,       /* stop grabbing and sending frames */
14      /* tcp client */
15      PACK_QUIT = 40,     /* end of call */
16  };
17
18
19  void netcSetLocalPort(const char *port);
20
21  void netcInit(void);
22  void netcFinish(void);
23
24  void netcStartGrab(void);
25  void netcStopGrab(void);
26  int  netcCameraAvailable(void);
27
28  #endif
```

netcamera.c

```
1   /* $Id$ */
2   /*************************************************************************
3    *
4    *  FILE            netcamera.c
5    *  MODULE OF       vidproxy
6    *
7    *  DESCRIPTION     Communication with camera server (c-grabber).
8    *
9    *  WRITTEN BY      Sverre H. Huseby <sverrehu@ifi.uio.no>
10   *
11   *************************************************************************/
12
13  #include <stdlib.h>
14  #include <stdio.h>
15  #include <string.h>
16  #include <unistd.h>
17
18  #include <shhmsg.h>
19  #include <shhnet.h>
20
21  #include "shhsched.h"
22  #include "netviewer.h"
23  #include "netcamera.h"
24
25  /*************************************************************************
26   *                                                                       *
```

```
27   *                     P R I V A T E   D A T A                      *
28   *                                                                  *
29   ********************************************************************/
30
31  static unsigned char netcInPacket[MAX_PACKET];
32
33  static char netcLocalTcpPort[81] = "";
34  static int  netcLocalTcpInitial = -1;
35  static int  netcLocalTcp = -1;
36  static int  netcLocalUdp = -1;
37
38  static SchedId netcAcceptReadyId;
39  static SchedId netcTcpReadyId;
40  static SchedId netcUdpReadyId;
41
42
43
44  /*******************************************************************
45   *                                                                 *
46   *                 P R I V A T E   F U N C T I O N S               *
47   *                                                                 *
48   *******************************************************************/
49
50  static void
51  netcCloseConnection(void)
52  {
53      if (netcLocalTcp >= 0) {
54          schedRemoveDesc(netcTcpReadyId);
55          netcTcpReadyId = 0;
56          tcpClose(netcLocalTcp);
57          netcLocalTcp = -1;
58      }
59      if (netcLocalUdp >= 0) {
60          schedRemoveDesc(netcUdpReadyId);
61          netcUdpReadyId = 0;
62          udpClose(netcLocalUdp);
63          netcLocalUdp = -1;
64      }
65  }
66
67  static void
68  netcUdpInput(void *clientData, int fd)
69  {
70      int n;
71
72      udpRead(fd, netcInPacket, sizeof(netcInPacket), &n);
73      netvSendPacket(netcInPacket, n);
74  }
75
76  static void
77  netcTcpInput(void *clientData, int fd)
78  {
79      unsigned char buff[128];
80      int           n, type;
81
82      if (tcpRead(fd, buff, sizeof(buff), &n) != SN_OK || n == 0) {
83          msgError("got error or empty packet, assuming broken connection\n");
84          netcCloseConnection();
85          return;
86      }
```

```
87      type = buff[0];
88      switch (type) {
89        case PACK_QUIT:
90          msgMessage("shutting down connection on request\n");
91          netcCloseConnection();
92          break;
93        default:
94          msgError("unknown packet received (%u)\n", (unsigned) type);
95      }
96  }
97
98  static void
99  netcAcceptConnection(void *clientData, int fd)
100 {
101     int           sock;
102     unsigned char buff[3];
103     char          name[100];
104     SNAddr        addr;
105     SNPort        port;
106
107     tcpAccept(fd, &sock);
108     tcpLinger(sock, 1);
109     if (netcLocalTcp >= 0) {
110         buff[0] = PACK_HAVE_FEED;
111         tcpWriteAll(sock, buff, 1);
112         tcpClose(sock);
113     } else {
114         inGetPeerAddrPort(sock, &addr, &port);
115         inHostAddrToName(addr, name, sizeof(name));
116         msgMessage("camera daemon connected from %s\n", name);
117         netcLocalTcp = sock;
118         netcTcpReadyId
119             = schedAddDesc(SCHED_READ_READY, sock, netcTcpInput, NULL);
120         udpClientSock(&netcLocalUdp);
121         netcUdpReadyId
122             = schedAddDesc(SCHED_READ_READY, netcLocalUdp, netcUdpInput, NULL);
123         inGetSockAddrPort(netcLocalUdp, &addr, &port);
124         buff[0] = PACK_WELCOME_PORT;
125         buff[1] = (port >> 8) & 0xFF;
126         buff[2] = port & 0xFF;
127         tcpWriteAll(sock, buff, 3);
128         if (netvGetNumConnections() > 0)
129             netcStartGrab();
130     }
131 }
132
133
134
135 /************************************************************************
136  *                                                                      *
137  *                  P U B L I C   F U N C T I O N S                     *
138  *                                                                      *
139  ************************************************************************/
140
141 void
142 netcSetLocalPort(const char *port)
143 {
144     strcpy(netcLocalTcpPort, port);
145 }
146
```

```
147 void
148 netcInit(void)
149 {
150     if (*netcLocalTcpPort == '\0')
151         sprintf(netcLocalTcpPort, "%d", DEFAULT_CAMERA_PORT);
152 
153     msgMessage("setting up tcp-server for camera daemon at port %s\n",
154                netcLocalTcpPort);
155     tcpServerOpen(netcLocalTcpPort, &netcLocalTcpInitial);
156     netcAcceptReadyId
157         = schedAddDesc(SCHED_READ_READY, netcLocalTcpInitial,
158                        netcAcceptConnection, NULL);
159     snSetErrorHandling(SN_REPORT_ERROR_AND_RETURN, NULL);
160 }
161 
162 void
163 netcFinish(void)
164 {
165     /* TODO: netcStopGrab(); */
166     netcCloseConnection();
167     if (netcLocalTcpInitial >= 0) {
168         schedRemoveDesc(netcAcceptReadyId);
169         tcpClose(netcLocalTcpInitial);
170         netcLocalTcpInitial = -1;
171     }
172 }
173 
174 void
175 netcStartGrab(void)
176 {
177     unsigned char buff[1];
178 
179     if (netcLocalTcp < 0)
180         return;
181     msgMessage("telling grabber to start grabbing\n");
182     buff[0] = PACK_START_GRAB;
183     tcpWriteAll(netcLocalTcp, buff, 1);
184 }
185 
186 void
187 netcStopGrab(void)
188 {
189     unsigned char buff[1];
190 
191     if (netcLocalTcp < 0)
192         return;
193     msgMessage("telling grabber to stop grabbing\n");
194     buff[0] = PACK_STOP_GRAB;
195     tcpWriteAll(netcLocalTcp, buff, 1);
196 }
197 
198 int
199 netcCameraAvailable(void)
200 {
201     return (netcLocalTcp >= 0);
202 }
```

netviewer.h

```
1   /* $Id$ */
2   #ifndef NETVIEWER_H
3   #define NETVIEWER_H
4
5   #include "network.h"
6
7   /* packet types */
8   enum {
9       /* tcp server */
10      PACK_NO_CAM = 0,      /* no camera available */
11      PACK_BUSY,            /* unable to serve you */
12      PACK_SEND_PORT,       /* send UDP port number */
13      PACK_WELCOME,         /* call accepted */
14      PACK_GO_AWAY,         /* call not accepted */
15      /* tcp client */
16      PACK_PORT = 10,
17      PACK_HANGUP,          /* end of call */
18  };
19  enum {
20      /* udp */
21      PACK_BLOCK = 20,      /* a block of an image */
22  };
23
24
25  void netvSetLocalPort(const char *port);
26
27  void netvInit(void);
28  void netvFinish(void);
29
30  void netvSendPacket(const unsigned char *packet, int len);
31  int  netvGetNumConnections(void);
32
33  #endif
```

netviewer.c

```
1   /* $Id$ */
2   /*************************************************************************
3    *
4    *  FILE            netviewer.c
5    *  MODULE OF       vidproxy
6    *
7    *  DESCRIPTION     Communication with viewer clients (c-client and
8    *                  SHHVid.java)
9    *
10   *  WRITTEN BY      Sverre H. Huseby <sverrehu@ifi.uio.no>
11   *
12   *************************************************************************/
13
14  #include <stdlib.h>
15  #include <stdio.h>
16  #include <string.h>
17  #include <unistd.h>
18
19  #include <shhmsg.h>
20  #include <shhnet.h>
21
```

```
22  #include "shhsched.h"
23  #include "netcamera.h"
24  #include "netviewer.h"
25
26  #undef COUNT_PACKETS
27
28  /*****************************************************************************
29   *                                                                           *
30   *                      P R I V A T E   D A T A                              *
31   *                                                                           *
32   *****************************************************************************/
33
34  #define MAX_CONNECTIONS 50
35
36  typedef struct {
37      int     tcpSock;
38      SchedId tcpSchedId;
39      int     udpSock;
40      SNAddr  udpAddr;
41      SNPort  udpPort;
42  #ifdef COUNT_PACKETS
43      unsigned long packetsSent;
44  #endif
45  } Connection;
46
47  static Connection conn[MAX_CONNECTIONS];
48  static int numConn = 0;
49
50  static char netvLocalTcpPort[81] = "";
51  static int  netvLocalTcpInitial = -1;
52
53  static SchedId netvAcceptReadyId;
54
55
56
57  /*****************************************************************************
58   *                                                                           *
59   *                  P R I V A T E   F U N C T I O N S                        *
60   *                                                                           *
61   *****************************************************************************/
62
63  static void netvTcpInput(void *clientData, int fd);
64
65  static int
66  netvAddConnection(int tcpSock, int udpSock)
67  {
68      Connection *c;
69
70      if (numConn == MAX_CONNECTIONS) {
71          msgError("out of space for more connections\n");
72          return -1;
73      }
74      c = &conn[numConn];
75  #ifdef COUNT_PACKETS
76      c->packetsSent = 0;
77  #endif
78      c->tcpSock = tcpSock;
79      c->udpSock = udpSock;
80      c->tcpSchedId
81          = schedAddDesc(SCHED_READ_READY, tcpSock, netvTcpInput, NULL);
```

```
82      if (++numConn == 1) {
83          msgMessage("first connection, setting up grabber\n");
84          netcStartGrab();
85      }
86      return numConn - 1;
87  }
88
89  static void
90  netvRemoveConnection(int tcpSock)
91  {
92      int        q;
93      Connection *c;
94
95      for (q = 0; q < numConn; q++)
96          if (conn[q].tcpSock == tcpSock)
97              break;
98      if (q == numConn)
99          return;
100     c = &conn[q];
101     schedRemoveDesc(c->tcpSchedId);
102     if (c->tcpSock >= 0)
103         tcpClose(c->tcpSock);
104     if (c->udpSock >= 0)
105         udpClose(c->udpSock);
106     memcpy(c, &conn[numConn - 1], sizeof(Connection));
107     if (--numConn == 0) {
108         msgMessage("no connections left, stopping grabber\n");
109         netcStopGrab();
110     }
111 }
112
113 static int
114 netvFindConnection(int tcpSock)
115 {
116     int q;
117
118     for (q = 0; q < numConn; q++)
119         if (conn[q].tcpSock == tcpSock)
120             return q;
121     return -1;
122 }
123
124 static void
125 netvTcpInput(void *clientData, int fd)
126 {
127     unsigned char buff[128];
128     char          name[100];
129     int           n, type, idx;
130     SNPort        port;
131
132     if (tcpRead(fd, buff, sizeof(buff), &n) != SN_OK || n == 0) {
133         msgError("got error or empty packet, assuming broken connection\n");
134         netvRemoveConnection(fd);
135         return;
136     }
137     idx = netvFindConnection(fd);
138     type = buff[0];
139     switch (type) {
140       case PACK_PORT:
141         inGetPeerStr(fd, name, sizeof(name), 1);
```

```
142         port = ((int) buff[1] << 8) | buff[2];
143         conn[idx].udpPort = port;
144         udpClientSock(&(conn[idx].udpSock));
145         msgMessage("incoming call from %s using UDP port %u\n",
146                    name, (unsigned) port);
147         buff[0] = PACK_WELCOME;
148         tcpWriteAll(fd, buff, 1);
149         break;
150       case PACK_HANGUP:
151         msgMessage("shutting down connection on request\n");
152         netvRemoveConnection(fd);
153         break;
154       default:
155         msgError("unknown packet received (%u)\n", (unsigned) type);
156     }
157 }
158
159 static void
160 netvAcceptConnection(void *clientData, int fd)
161 {
162     int           sock, idx;
163     unsigned char buff[1];
164     SNAddr        addr;
165     SNPort        port;
166
167     tcpAccept(fd, &sock);
168     tcpLinger(sock, 1);
169     inGetPeerAddrPort(sock, &addr, &port);
170     if (!netcCameraAvailable()) {
171         buff[0] = PACK_NO_CAM;
172         tcpWriteAll(sock, buff, 1);
173     } else if ((idx = netvAddConnection(sock, -1)) < 0) {
174         buff[0] = PACK_BUSY;
175         tcpWriteAll(sock, buff, 1);
176     } else {
177         conn[idx].udpAddr = addr;
178         conn[idx].tcpSchedId
179             = schedAddDesc(SCHED_READ_READY, sock, netvTcpInput, NULL);
180         buff[0] = PACK_SEND_PORT;
181         tcpWriteAll(sock, buff, 1);
182     }
183 }
184
185
186
187 /*************************************************************************
188  *                                                                       *
189  *                  P U B L I C   F U N C T I O N S                      *
190  *                                                                       *
191  *************************************************************************/
192
193 void
194 netvSetLocalPort(const char *port)
195 {
196     strcpy(netvLocalTcpPort, port);
197 }
198
199 void
200 netvInit(void)
201 {
```

```
202     if (*netvLocalTcpPort == '\0')
203         sprintf(netvLocalTcpPort, "%d", DEFAULT_VIEWER_PORT);
204 
205     msgMessage("setting up tcp-server for viewers at port %s\n",
206                netvLocalTcpPort);
207     tcpServerOpen(netvLocalTcpPort, &netvLocalTcpInitial);
208     netvAcceptReadyId
209         = schedAddDesc(SCHED_READ_READY, netvLocalTcpInitial,
210                        netvAcceptConnection, NULL);
211     snSetErrorHandling(SN_REPORT_ERROR_AND_RETURN, NULL);
212 }
213 
214 void
215 netvFinish(void)
216 {
217     int q;
218 
219     for (q = 0; q < numConn; q++) {
220         if (conn[q].tcpSock >= 0)
221             tcpClose(conn[q].tcpSock);
222         if (conn[q].udpSock >= 0)
223             tcpClose(conn[q].tcpSock);
224         schedRemoveDesc(conn[q].tcpSchedId);
225     }
226     numConn = 0;
227     if (netvLocalTcpInitial >= 0) {
228         tcpClose(netvLocalTcpInitial);
229         netvLocalTcpInitial = -1;
230         schedRemoveDesc(netvAcceptReadyId);
231     }
232 }
233 
234 void
235 netvSendPacket(const unsigned char *packet, int len)
236 {
237     int        q, n;
238     Connection *c;
239 #ifdef COUNT_PACKETS
240     static int delay = 0;
241     char name[81];
242 #endif
243 
244     if (!len)
245         return;
246     for (q = 0; q < numConn; q++) {
247         c = &conn[q];
248         if (c->udpSock >= 0) {
249             udpWriteTo(c->udpSock, packet, len, c->udpAddr, c->udpPort, &n);
250             if (n < len)
251                 msgError("partial packet sent\n");
252         }
253 #ifdef COUNT_PACKETS
254         ++(c->packetsSent);
255 #endif
256     }
257 #ifdef COUNT_PACKETS
258     if (++delay == 100) {
259         delay = 0;
260         for (q = 0; q < numConn; q++) {
261             c = &conn[q];
```

```
262             inHostAddrToName(c->udpAddr, name, sizeof(name));
263             printf("sent %ld packets to %s\n", c->packetsSent, name);
264         }
265     }
266 #endif
267 }
268
269 int
270 netvGetNumConnections(void)
271 {
272     return numConn;
273 }
```

shhsched.h

Identical to the file used in the grabber. Please refer to page 94 for full listing.

shhsched.c

Identical to the file used in the grabber. Please refer to page 95 for full listing.

Appendix E

Recoding MPEG to JPEG and GIF

This appendix explains the method used for converting MPEG streams to GIF and JPEG files in chapter 2.5 on page 14. After shortly describing the tools used, figure E.1 presents the script used to do the actual conversion.

The MPEG streams were decoded using Andy Hung's MPEG codec[1], which translates from a single MPEG file, to three files for each frame of the movie. The three files, given extensions `.Y`, `.U` and `.V`, contains the three components from the YUV (or actually YCbCr) color space of the frame in question.

PPM (Portable PixMap file format) was chosen as an intermediate format during conversion, since fully command line driven support programs are available, simplifying conversion of a large number of files. A large collection of support programs, known as *netpbm*[2], provides conversion between many popular image file formats, in addition to elementary image processing.

Matching YUV-files were combined into a single PPM-file using `cyuv2ppm`[3], also written by Andy Hung.

Conversion from PPM to JPEG was done with Independent JPEG Group's program `cjpeg`, found in the package `jpegsrc.v6a.tar.gz`[4].

GIF-conversion was performed in two steps. The first step quantized the image to 256 colors, the maximum number of colors in a GIF image. Quantization was done using `ppmquant`. The second step converted the quantized PPM image to GIF using `ppmtogif`. Both programs can be found in the above mentioned netpbm package.

The shell script in the following figure, decodes a single MPEG file, given it's name and horizontal and vertical frame sizes. After decoding to a number of YUV-files, the conversion to JPEG and GIF is done, followed by removal of all intermediate files. Note that `cjpeg` defaults to a quality of 75. The quality settings range from 0 (worst)

[1] ftp://havefun.stanford.edu/pub/mpeg/MPEGv1.2.2.tar.Z
[2] ftp://wuarchive.wustl.edu/graphics/graphics/packages/NetPBM/
[3] ftp://havefun.stanford.edu/pub/cv/VCv1.2.2.tar.Z
[4] ftp://ftp.uu.net/graphics/jpeg/jpegsrc.v6a.tar.gz

to 100 (best), and the number given is used to scale the quantization tables.

```
#!/bin/sh

# NAME=bart-temple
# WIDTH=192
# HEIGHT=144

NAME=bjork
WIDTH=160
HEIGHT=120

# NAME=enterprise
# WIDTH=176
# HEIGHT=144

mkdir -p decode/$NAME
cd decode/$NAME
mpeg -d -s ../../$NAME.mpg $NAME.
for file in *.Y
do
    base=‘basename $file .Y‘
    cyuv2ppm $base $base.ppm -iw $WIDTH -ih $HEIGHT
    cjpeg $base.ppm > $base.jpg
    ppmquant 256 $base.ppm | ppmtogif > $base.gif
    rm $base.[YUV] $base.ppm
done
```

Figure E.1: Shell-script used to recode the MPEG files to JPEG and GIF.

Appendix F

Internet Links

This appendix provides links to relevant information found on the World Wide Web. Note that more links may be found in the bibliography starting at page 125.

Video Applications for WWW

This section contains links to existing programs for inlining video in Web browsers. The following Video on Demand -like programs are available for playback of movie files:

- *"Action"*
  http://open2u.com/action/action.html
  An MPEG video and sound player plug-in for Windows95.
- *"Apple QuickTime Plug-In"*
  http://quicktime.apple.com/
  Plug-in playing QuickTime movies on Macintosh and Windows.
- *"InterVU"*
  http://www.intervu.com/player/aboutpre.html
  MPEG-playing plug-in for Windows95 and Macintosh.
- *"Plug-in Plaza, Multimedia"*
  http://browserwatch.iworld.com/plug-in-mm.html
  Lists lots of known plug-ins for multimedia applications.

MBone

The following links provide information on MBone, the multicasting backbone.

- *"Index to MBone information"*
  http://www.mang.canterbury.ac.nz/~busa057/mbone/
  Pointers to MBone information in general, and precompiled programs in particular.
- *"The MBone FAQ"*
  http://www.best.com/~prince/techinfo/mbone.faq.html
  Contains a Frequently Asked Questions -list for the MBone.

- *"The MBone Information Web"*
  http://www.best.com/~prince/techinfo/mbone.html
  Contains pointers and information.

Video Compression

Pointers to resources describing various video compression standards and applications.

- *"The MPEG Meta-guide"*
  http://www.mpeg.org/
  "Is the most complete, comprehensive and up-to-date index of MPEG resources on the Web."

- *"MPEG Archive"*
  http://www.powerweb.de/mpeg/
  Contains information, links, software and movies.

- *"MPEG Information Page"*
  http://www.vol.it/MPEG/
  Contains answers to frequently asked questions.

- *"MPEG Research at U.C. Berkeley"*
  http://bmrc.berkeley.edu/projects/mpeg/
  Describes the MPEG encoder and decoder developed at Berkeley.

- *"MPEG Overview"*
  http://www.c-cube.com/tecno/mpeg.html
  Gives a brief and informative overview of the MPEG standard.

- *"PVRG-MPEG"*
  ftp://havefun.stanford.edu/pub/mpeg/
  A source code package for MPEG encoding/decoding, developed by the Portable Video Research Group at Stanford University.

- *"PVRG-P64"*
  ftp://havefun.stanford.edu/pub/p64/
  Source code package for H.261 encoding/decoding, developed by the Portable Video Research Group at Stanford University.

Image Compression

Information regarding single image compression.

- *"JPEG Frequently Asked Questions"*
  ftp://rtfm.mit.edu/pub/usenet/news.answers/jpeg-faq/

- *"PNG (Portable Network Graphics) Home Page"*
  http://www.wco.com/~png/
  Lots of PNG-related pointers. PNG is supposed to be the successor of GIF.

- *"PVRG-JPEG"*
  ftp://havefun.stanford.edu/pub/jpeg/
  Source code package for JPEG encoding/decoding, developed by the Portable Video Research Group at Stanford University.

- ftp://ftp.ncsa.uiuc.edu/misc/file.formats/graphics.formats/
  Contains files describing various image file formats.

Research Projects

Research project on transferring video on the Internet.

- *"The LAVA Project"*
  http://www.nr.no/lava/
  is led by Telenor Research & development The main focus of the LAVA project is the delivery of video using ATM technology. LAVA is a Norwegian acronym for "Delivery of video over ATM".

- *"MERCI"*
  http://www-ks.rus.uni-stuttgart.de/PROJ/MERCI/
  MERCI is short for "Multimedia European Research Conferencing Integration". The objective of the project is to provide all the technology components, other than the data network itself, to allow proper deployment of the tools for European multimedia collaboration in Europe.

Miscellaneous

Items that didn't fit elsewhere.

- *"Internet Documentation (RFC's, FYI's, etc.) and IETF Information"*
  http://ds.internic.net/ds/dspg0intdoc.html
  Contains all RFCs and other documents describing the Internet standards and draft standards.

Bibliography

[1] W. Richard Stevens. *Unix Network Programming.* Prentice Hall Software Series, 1990.

[2] E. Krol and E. Hoffman. FYI on "What is the Internet?", 1993. RFC 1462, available at http://ds.internic.net/rfc/rfc1462.txt.

[3] J. Reynolds and J. Postel. The request for comments reference guide, 1987. RFC 1000, available at http://ds.internic.net/rfc/rfc1000.txt.

[4] Bruce Sterling. Short history of the Internet, 1993. available at gopher://gopher.isoc.org:70/00/internet/history/short.history.of.internet.

[5] Robert H Zakon. Hobbes' Internet timeline v2.5, 1996. available at http://info.isoc.org/guest/zakon/Internet/History/HIT.html.

[6] T. Berners-Lee and R. Cailliau. WorldWideWeb: Proposal for a HyperText project, 1990. available at http://www.w3.org/pub/WWW/Proposal.html.

[7] Robert Cailliau. A little history of the world wide web, 1995. available at http://www.w3.org/pub/WWW/History.html.

[8] Andy C. Hung. *PVRG-P64 CODEC 1.1*, 1993. available at ftp://havefun.stanford.edu/pub/p64/.

[9] Foley, van Damme, Feiner, and Huhges. *Computer Graphics: Principles and Practice.* Addison-Wesley Publishing Company, second edition, 1990.

[10] William K. Pratt. *Digital Image Processing.* Wiley & sons Inc., second edition, 1991.

[11] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing.* Addison–Wesley Publishing Company, Inc., 1992.

[12] Majid Rabbani and Paul W. Jones. *Digital Image Compression Techniques.* SPIE Optical Engineering Press, 1991.

[13] CompuServe Incorporated. *Graphics Interchange Format, Version 87a*, 1987. available at ftp://ftp.ncsa.uiuc.edu/misc/file.formats/graphics.formats/gif87a.doc.

[14] CompuServe Incorporated. *Graphics Interchange Format, Version 89a*, 1989. available at ftp://ftp.ncsa.uiuc.edu/misc/file.formats/graphics.formats/gif89a.doc.

[15] William B. Pennebaker and Joan L. Mitchell. *JPEG Still Image Data Compression Standard (incl. ISO DIS 10918-1/2).* Van Nostrad Reinhold, 1993.

[16] Gregory K. Wallace. The JPEG still picture compression standard. *Communications of the ACM*, 1991.

[17] Tom Lane et al. JPEG frequently asked questions, October 1996. available at ftp://rtfm.mit.edu/pub/usenet/news.answers/jpeg-faq/.

[18] Eric Hamilton. JPEG file interchange format version 1.02. Technical report, C-Cube Microsystems, 1992. available at ftp://ftp.uu.net/graphics/jpeg/jfif.ps.gz.

[19] Frank Gadegast et al. The MPEG-FAQ version 4.1, June 1996. available at ftp://rtfm.mit.edu/pub/usenet/news.answers/mpeg-faq/.

[20] International Telecommunication Union, Telecommunication Standardization Sector (ITU-T). *ITU-T Recommendation H.261: Line Transmission of Non-Telephone Signals — Video Codec for Audiovisual Services at $p \times 64$ kbits*, March 1993.

[21] International Telecommunication Union, Telecommunication Standardization Sector (ITU-T). *ITU-T Draft Recommendation H.263: Line Transmission of Non-Telephone Signals — Video Coding for Low Bitrate Communication*, May 1996.

[22] C-Cube Microsystems. Compression technology — MPEG overview. available at http://www.c-cube.com/tecno/mpeg.html.

[23] Didier Le Gall. MPEG: A video compression standard for multimedia applications. *Communications of the ACM*, vol. 34(no. 4), April 1991.

[24] Lenoardo Chiariglione. The development of an integrated audiovisual coding standard: MPEG. *Proceedings of the IEEE*, vol. 83:151–157, February 1995.

[25] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols.* Addison-Wesley Publishing Company, 1994.

[26] Fred Halsall. *Data Communications, Computer Networks and Open Systems.* Addison-Wesley, third edition, 1994.

[27] J. Mogul and J. Postel. Internet standard subnetting procedure, 1985. RFC 950, available at http://ds.internic.net/rfc/rfc950.txt.

[28] P. Mockapetris. Domain names — concepts and facilities, 1987. RFC 1034, available at http://ds.internic.net/rfc/rfc1034.txt.

[29] P. Mockapetris. Domain names — implementation and specification, 1987. RFC 1035, available at http://ds.internic.net/rfc/rfc1035.txt.

[30] Jon Postel. Transmission control protocol, 1981. RFC 793, available at http://ds.internic.net/rfc/rfc793.txt.

[31] Jon Postel. User datagram protocol, 1980. RFC 768, available at http://ds.internic.net/rfc/rfc768.txt.

[32] J. Reinolds and J. Postel. Assigned numbers, 1994. RFC 1700, available at http://ds.internic.net/rfc/rfc1700.txt.

[33] J. Postel and J. Reynolds. File transfer protocol (FTP), 1985. RFC 959, available at http://ds.internic.net/rfc/rfc959.txt.

[34] Jonathan B. Postel. Simple mail transfer protocol, 1982. RFC 821, available at http://ds.internic.net/rfc/rfc821.txt.

[35] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol – HTTP/1.0, 1996. RFC 1945, available at http://ds.internic.net/rfc/rfc1945.txt.

[36] Randi Haraldsø. Distribusjon av multimedia. In *Nye teknologier — Multimedia 7. november*. Den Norske Dataforening, 1996.

[37] Jeffrey Mogul. Broadcasting Internet datagrams in the presence of subnets, 1984. RFC 922, available at http://ds.internic.net/rfc/rfc922.txt.

[38] David C. Plummer. An ethernet address resolution protocol, or converting network protocol addresses to 48.bit ethernet address for transmission on ethernet hardware, 1982. RFC 826, available at http://ds.internic.net/rfc/rfc826.txt.

[39] R. Finlayson, T. Mann, J. Mogul, and M. Theimer. A reverse address resolution protocol, 1984. RFC 903, available at http://ds.internic.net/rfc/rfc903.txt.

[40] S. Armstrong, A. Freier, and K. Marzullo. Multicast transport protocol, 1992. RFC 1301, available at http://ds.internic.net/rfc/rfc1301.txt.

[41] R. Braudes and S. Zabele. Requirements for multicast protocols, 1993. RFC 1458, available at http://ds.internic.net/rfc/rfc1458.txt.

[42] S. Deering. Host extensions for IP multicasting, 1989. RFC 1112, available at http://ds.internic.net/rfc/rfc1112.txt.

[43] Michael R. Macedonia and Donald P. Brutzman. MBONE provides audio and video across the Internet. *IEEE Computer*, vol. 2Z(no. 4):30–36, April 1994. available at ftp://taurus.cs.nps.navy.mil/pub/mbmg/mbone.ps.

[44] Hans Eriksson. MBONE: The multicast backbone. *Communications of the ACM*, vol. 37(no. 8), August 1994. available at http://www.mang.canterbury.ac.nz/~busa057/mbone/art1.html.

[45] Stephen Casner and Stephen Deering. First IETF Internet audiocast. *ACM Sigcomm Computer Communication Review*, vol. 22(no. 23):92–97, July 1992. available at ftp://venera.isi.edu/pub/ietf-audiocast.article.ps.

[46] Stephen Casner. Are you on the MBone? *IEEE Multimedia*, vol. 1(no. 2):76–79, 1994. available at http://www.mang.canterbury.ac.nz/~busa057/mbone/art2.html.

[47] Mark Handley and Van Jacobson. SDP: Session description protocol, 1996. Internet draft, currently available at http://ds.internic.net/internet-drafts/draft-ietf-mmusic-sdp-02.txt.

[48] Mark Handley. SAP: Session announcement protocol, 1996. Internet draft, currently available at http://ds.internic.net/internet-drafts/draft-ietf-mmusic-sap-00.txt.

[49] M. Handley, H. Schulzrinne, and E. Schooler. SIP: Session invitation protocol, 1996. Internet draft, currently available at http://ds.internic.net/internet-drafts/draft-ietf-mmusic-sip-01.txt.

[50] Tim Berners-Lee. Worldwide web seminar, 1992. available at http://www.w3.org/pub/WWW/Talks/General.html.

[51] N. Borenstein, Bellcore, and N. Freed. MIME (multipurpose Internet mail extensions) part one: Mechanisms for specifying and describing the format of internet message bodies, 1993. RFC 1521, available at http://ds.internic.net/rfc/rfc1521.txt.

[52] The common gateway interface. available at http://hoohoo.ncsa.uiuc.edu/cgi/.

[53] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A transport protocol for real-time applications, 1996. RFC 1889, available at http://ds.internic.net/rfc/rfc1889.txt.

[54] H. Schulzrinne. RTP profile for audio and video conferences with minimal control, 1996. RFC 1890, available at http://ds.internic.net/rfc/rfc1890.txt.

[55] D. Hoffman, G. Fernando, and V. Goyal. RTP payload format for MPEG1/MPEG2 video, 1996. RFC 2038, available at http://ds.internic.net/rfc/rfc2038.txt.

[56] T. Turletti and C. Huitema. RTP payload format for H.261 video streams, 1996. RFC 2032, available at http://ds.internic.net/rfc/rfc2032.txt.

[57] L. Berc, W. Fenner, R. Frederick, and S. McCanne. RTP payload format for JPEG-compressed video, 1996. RFC 2035, available at http://ds.internic.net/rfc/rfc2035.txt.

[58] Tim Dorcey. CU-SeeMe desktop videoconferencing software. *Connexions*, vol. 9(no. 3), March 1995. available at http://cu-seeme.cornell.edu/DorceyConnexions.html.

[59] Michel Carleer. *CU-SeeMe (tm) for Windows*, March 1996. Available at ftp://cu-seeme.cornell.edu/pub/video/PC.CU-SeeMeCurrent/readme.txt.

[60] T. Berners-Lee and D. Connolly. Hypertext markup language – 2.0, 1995. RFC 1866, available at http://ds.internic.net/rfc/rfc1866.txt.

[61] D. Raggett and D. Connolly. Introducing HTML 3.2, 1996. available at http://www.w3.org/pub/WWW/MarkUp/Wilbur/.

[62] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform resource locators (URL), 1994. RFC 1738, available at http://ds.internic.net/rfc/rfc1738.txt.

[63] Brian Kantor and Phil Lapsley. Network news transfer protocol, 1986. RFC 977, available at http://ds.internic.net/rfc/rfc977.txt.

[64] F. Anklesaria, M. McCahill, P. Lindner, D. Johnson, D. Torrey, and B. Alberti. The Internet gopher protocol, 1993. RFC 1436, available at http://ds.internic.net/rfc/rfc1436.txt.

[65] Netscape Communications Corporation. *An Exploration of Dynamic Objects.* available at http://home.netscape.com/assist/net_sites/pushpull.html.

[66] Michael Swafford and Dane Dwyer. CS397KC final project presentation, 1995. available at http://yertle.csl.uiuc.edu/multimedia/java_mpeg/.

[67] Zhigang Chen, See-Mong Tan, Roy H. Campbell, and Yongcheng Li. Real time video and audio in the world wide web. Technical report, Vosaic Corp., 1995. available at http://choices.cs.uiuc.edu/Papers/New/vosaic/vosaic.html.

[68] Dave Garaffa. Browserwatch, 1996. http://browserwatch.iworld.com/.

[69] Netscape Communications Corporation. *Plug-in Guide.* available at http://home.netscape.com/eng/mozilla/3.0/handbook/plugins/pguide.htm.

[70] David Bank. The Java saga. *HotWired*, December 1995. available at http://www.hotwired.com/wired/3.12/features/java.saga.html.

[71] Jason English. It all started with an angry letter, 1996. available at http://www.javasoft.com/nav/whatis/index.html.

[72] Sun Microsystems Computer Corporation. *The Java Virtual Machine Specification*, 1.0 beta, draft edition, August 1995. available at http://www.javasoft.com/doc/language_vm_specification.html.

[73] James Gosling, Frank Yellin, and The Java Team. *Java API Documentation version 1.0.3*, 1996. available at http://www.javasoft.com/products/JDK/1.0.2/api/packages.html.

[74] Laura Lemay and Charles L. Perkins. *Teach Yourself Java in 21 Days.* Sams.net Publishing, first edition, 1996.

[75] W. Richard Stevens. *Advanced Programming in the Unix Environment.* Addison-Wesley Publishing Company, 1992.

[76] Andrew S. Tannenbaum. *Operating Systems: Design and Implementation.* Prentice Hall, 1987.

[77] Adrian Nye. *Xlib Programming Manual for Version 11.* O'Reilley & Associates, Inc., 1995.

[78] Adrian Nye, editor. *Xlib Reference Manual for Version 11.* O'Reilley & Associates, Inc., 1993.

[79] Carolyn Curtis. *IRIS Digital Media Programming Guide*, chapter 11–15. Silicon Graphics Inc., 1994.

[80] M. J. Usher. *Information Theory for Information Technologists.* Macmillan Computer Science Series, 1984.

[81] Ross N. Williams. *Adaptive Data Compression.* Kluwer Academic Publishers, 1991.

[82] Mark Nelson. *The Data Compression Book*. M&T Publishing, Inc., 1991.

[83] Frequently asked questions from comp.compression, 1994.

[84] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, vol. 23(no. 3):337–343, May 1977.

[85] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, vol. 24(no. 5):530–536, September 1978.