CS 595 / 596

Design of a Medical-Video-on-Demand System over ATM-Networks

Tilman Wolf
August 1997

This project is in partial fulfillments for the requirements for the degree Master of Science in Computer Science at Washington University in St. Louis.

1 Project Description and Goals 3
2 Background and Significance 4
2.1 Medical Background 4
2.2 Video Data 4
2.3 Networking 4
2.4 Video Server 5
3 System Outline 6
4 MARS Video Server 8
4.1 Design and Function of MARS 8
4.2 NetBSD Installation for MARS 11
4.3 MMX 11
4.4 Modifications for this Project 12
5 Video Acquisition 15
5.1 Video Capture Hardware 15
5.2 AVI File Format 15
5.3 AVI and M-JPEG 16
5.4 Data Preparation for Server 20
6 Video Playback 21
6.1 ATM under Microsoft Windows 21
6.2 Video for Windows 23
6.3 Playback Software 24
7 Performance 26
7.1 Overall performance 26
7.2 Performance Analysis 27
8 Summary and Future Work 30
8.1 Summary 30
8.2 Future Work 30
9 Appendix A: Hardware Specification 32
10 Appendix B: NetBSD Installation 33
10.1 Installing NetBSD release 33
10.2 Compiling NetBSD Kernel 36
10.3 Installing NetBSD current 36
11 Appendix C: MARS Installation 39
11.1 Activating MARS 39
11.2 Compiling MARS 40
11.3 Miscellaneous about MARS 40
12 Appendix D: Video Meta Data Software 42
13 Appendix E: Playback Software 46
13.1 JAVA-Applet to control MARS 46
13.2 Playback Format Translator 48
14 Appendix F: Useful URLs 54
15 Bibliography 55

1 Project Description and Goals

The goal of this project was to design and implement a system for acquisition, storage and retrieval of medical video data over an ATM network.

The system is based on the MARS [1] (Massively-parallel And Real-time Storage) project at Washington University. MARS allows the recording of full resolution, full frame rate video on a digital storage recording station. Video is acquired, compressed and ATMized using the MMX [2]. The data is transmitted to a central server for storage. Then video can be accessed remotely via a high bandwidth ATM-network. The playback can be controlled interactively.

The first goal of this project was to duplicate the MARS system. It has only been used experimentally so far. Therefore, a well documented installation routine had to be developed.

Another objective was to design a low-cost playback workstation in order to minimize hardware expenses. The workstation is Pentium-based and supported by Microsoft Windows for compatibility with the clinical workstations being developed by Project Spectrum [3]. Also software to access and control the new hardware had to be developed.

Finally the MARS system had to be adapted for the usage with the new hardware.

The performance of the resulting system was measured and improved where possible.

2 Background and Significance

2.1 Medical Background

Many forms of medical examinations produce video data. The most common techniques are ultrasound, fluoroscopy and endoscopy. It is desirable to acquire patient examination data at geographically distributed sites and forward the data to a central center of expertise for diagnostic review. Making exam data electronically available allows physicians to access the data via a network [4]. This resolves logistic problems that occur when only one copy of an examination result is available, but several physicians need to access this information at the same time.

2.2 Video Data

Real-time transmission of high quality digital video requires a large amount of bandwidth. For example, 640 x 480 pixel x 24 bits/pixel x 30 frames/second without compression requires bandwidths of up to 220 Mb/s. Even high-bandwidth networks cannot handle this easily. Therefore a lossy compression algorithm is to be used to reduce the bandwidth. A common compression method is Motion-JPEG. M-JPEG compresses each video frame independently with the JPEG algorithm defined by ISO 10918. Although there are methods, such as MPEG, that achieve a higher compression ratio, most commercial products use M-JPEG because MPEG requires complex and expensive hardware for the encoding procedure. Only M-JPEG encoded video is used in this project.

2.3 Networking

The data rate of a visually loss-less full-screen, full frame rate, M-JPEG compressed video is around 10-15 Mb/s. Considering the network has to be able to transmit several video streams at the same time, a bandwidth of several times 15 Mb/s is required. This excludes modems and low-speed LANs as network connections. High-speed LANs like Fast Ethernet are limited to short distances. Therefore ATM is the network standard of choice. It supports high bandwidth (155 Mb/s) and can be used as a LAN as well as a WAN.

2.4 Video Server

The Massively-parallel And Real-time Storage (MARS) server that was developed at Washington University is used for this project because it has many of the features that are required. It stores and plays high-quality video (640 x 480 pixels, 30 frames per second) over an ATM network. The video playback can be controlled interactively by the user. And MARS can be modified easily because the source codes are available.

Unfortunately MARS requires a MultiMedia eXplorer (MMX) device at each playback workstation. The current price for a MMX is around $10,000. This is several times the price of the workstation itself. To allow a wider acceptance of electronically distributed medical video it is essential to develop a system which decreases the decoder hardware costs. Especially the costs for playback workstations should be minimized, because they are numerous. The design proposed in this project requires only two off-the-shelf adapters: a video codec card and an ATM network interface card. Both adapters designed for the PCI bus interface are available for less than $1,000 each.

3 System Outline

A block diagram of the designed system is shown in Figure 1.

The video data is generated by a medical instrument or a video camera at a workstation. There the data is temporarily stored and prepared so it can be input to the MARS server. Then the data is sent to the server where it is stored. The server transmits the video data on request to the appropriate workstation. The workstation can control the video playback with Play, Pause, Rewind, Fast Forward, etc. commands. The MARS is able to serve several workstations with different video streams at the same time.

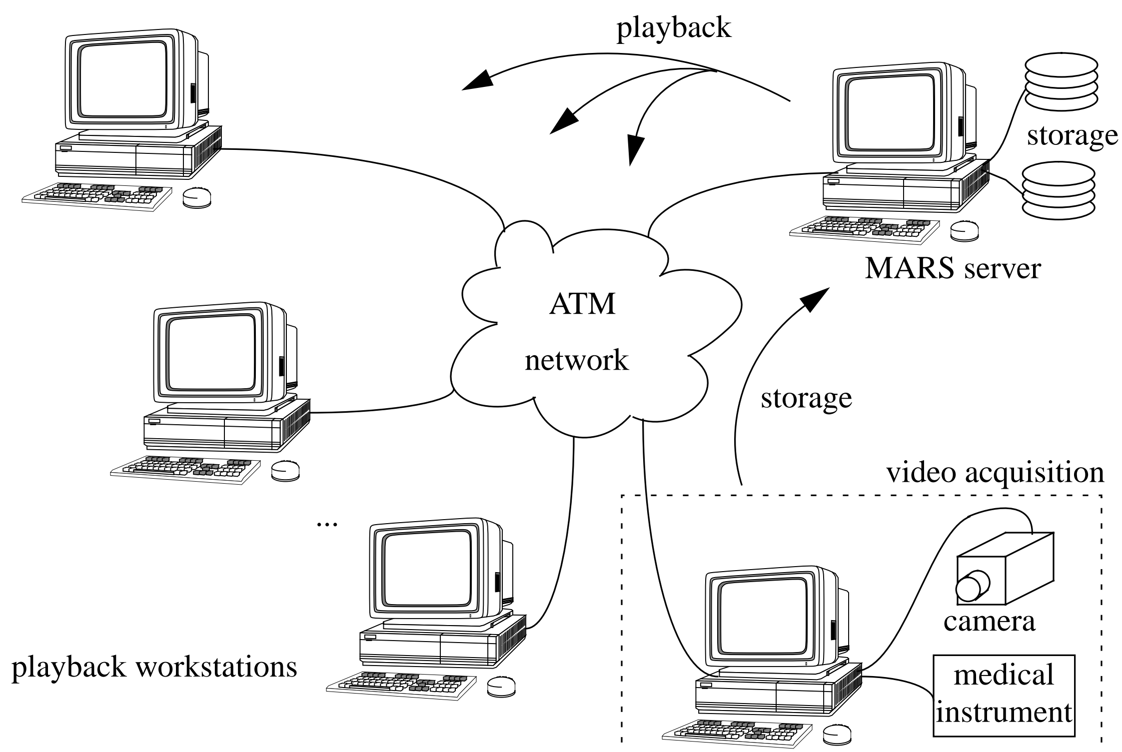

FIGURE 1. Block diagram of designed system.

Figure 2 shows the modification in the design of the playback workstations. The original playback system uses the MMX to decode the data stream from MARS and display it. The MMX is controlled by the workstation which also sends the commands to the MARS server.

In the modified playback scheme, video data is received by an ATM network adapter. Then the playback software modifies the data slightly so it can be displayed by the video codec board. The software also controls the playback on MARS.

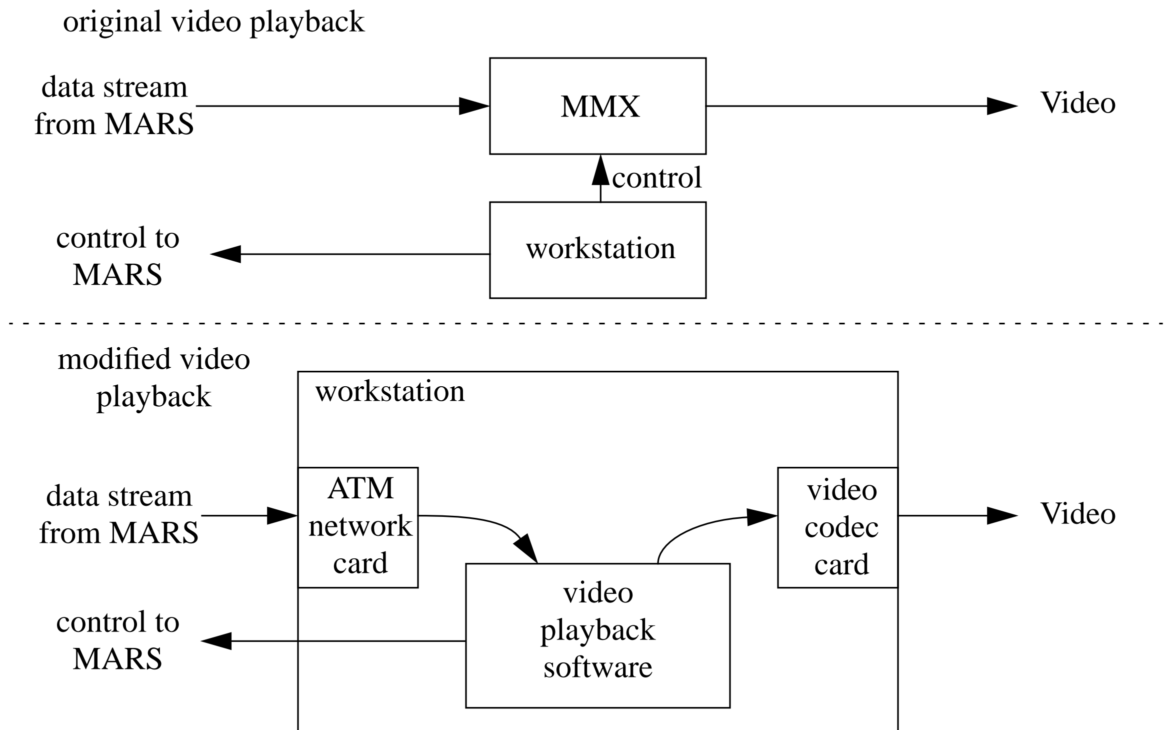

FIGURE 2. Comparison between original MMX-based and modified playback workstation

4 MARS Video Server

The architecture of a large scale multimedia-on-demand system is described in [1]. The main issue for a video server is to overcome the bottlenecks in I/O to storage and network. The MARS system used in this project can supply real-time video data to several clients, and it allows interactive control of the playback.

4.1 Design and Function of MARS

A block diagram of the current version of MARS is shown in Figure 3.

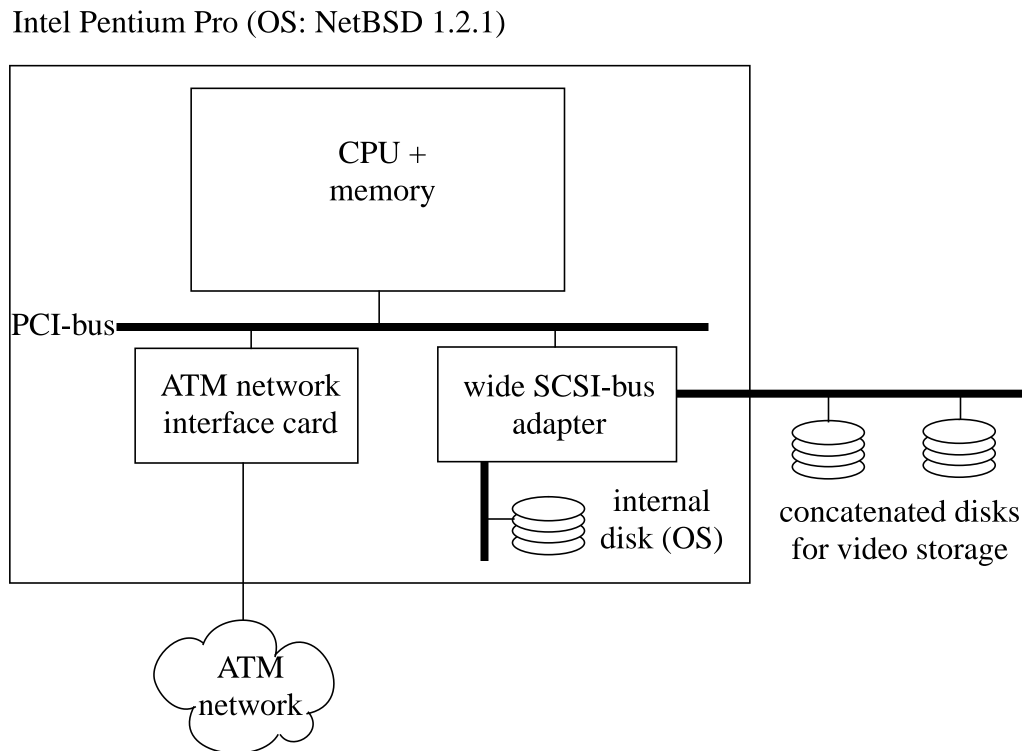

FIGURE 3. Block diagram of video server

4.1.1 Hardware

The MARS server is an Intel Pentium Pro processor based computer with a PCI-bus. The complete hardware specifications are included in Appendix A. The operating system is NetBSD, which was adjusted to support the Efficient Networks ATM card [5] and other

functions that are essential for the server like Real-Time Upcalls (RTUs) and Concatenated Disks (ccds) [6].

The storage I/O is done via two ultra wide SCSI busses. The one SCSI-bus is reserved for the operating system and 'normal' file access. The other is used for storage and retrieval of video data. The periodic access to the video storage disks must happen at the scheduled time. If video I/O is blocked by the operating system then the server cannot send the video data at the required constant frame rate. The operating system performs swapping and other I/O operations in a non-deterministic manner. Separating the operating system disk from video data disks decreases the influence of OS activities on the real-time performance of the server.

4.1.2 Software

The MARS server consists of the following software:

- WWW-server to access the stored videos
- Record daemon to record video

The WWW-server supplies the web pages describing stored videos. The server also controls the access to different user accounts, and controls the transmission of the video data over the network.

The recording software allows the user to store a new video at the server, view it and publish it on the World-Wide Web. The existing recording software uses the MMX, and is not considered any further here.

There is also software to extract single frames from video files and to manage the user accounts.

This software is written in C and can be compiled on the NetBSD platform. The installation of the software is described in Appendix C.

Figure 4 shows an example of how the server software operates.

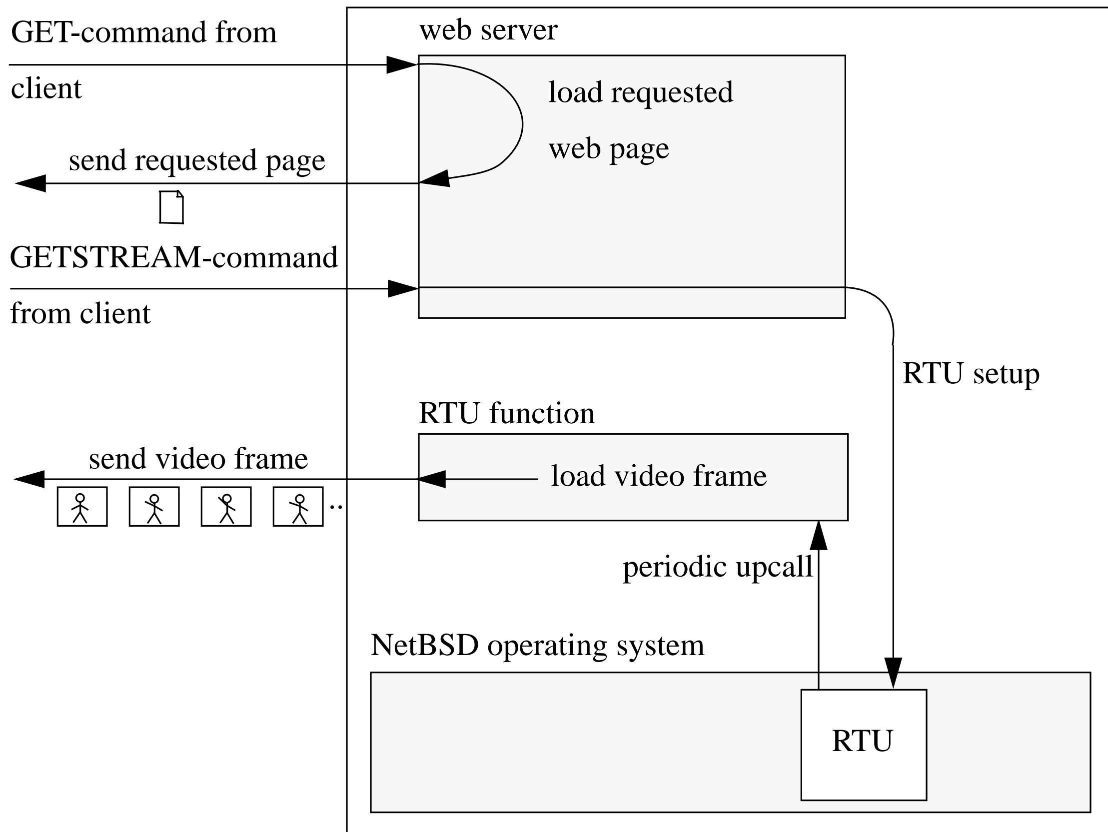

FIGURE 4. Example of server software operation

The MARS server sends the web pages that are requested with the `GET` command, like any other web server. It parses the request and extracts the file name from the command. Then it opens this file and sends its contents to the client. The following example shows a request that corresponds to the URL `http://server.domain/example.html` (`HTTP/1.0` describes which version of the HyperText Transfer Protocol is being used):

```
GET /example.html HTTP/1.0
```

For the Multimedia-on-Demand services there is the special `GETSTREAM` command. A `GETSTREAM` command contains information about which video is requested, if video and audio is requested, which protocol and which PVCs are to be used. The following example requests the video `example.mjpeg`:

```
GETSTREAM accs/username/example/example.mjpeg?BOTH
+HARDWARE+IMAGE=root/movie2.gif+RAWATM
+PROTO=AAL0+0+500+0+501+LOOPPLAY HTTP/1.0
```

On a `GETSTREAM` request, the server sets up a Real-Time Upcall function. The function that sends video frames onto the network is passed to the operating system. The operating system uses the built in Real-Time Upcall mechanism to activate this function at specified, periodic intervals. Every time the function is called it loads the next frame from disk and sends it out on the ATM network.

4.2 NetBSD Installation for MARS

The MARS server software was written for a NetBSD operating system. NetBSD is a freely available version of UNIX based in the Berkley Software Distribution. NetBSD is distributed in two versions. The 'release' version contains well tested software which can be expected to work. The 'current' version contains newly developed drivers and software which is still being tested and improved. The current version does not necessarily operate properly. In this project the current NetBSD is required because the driver for the Efficient Network ATM Adapter, Real-Time Upcalls and Concatenated Disks are not part of the latest release (NetBSD 1.2.1).

A complete installation guide for NetBSD can be found in Appendix B.

4.3 MMX

The MultiMedia eXplorer (MMX) is a host-independent device that can transmit and receive full resolution, full frame rate M-JPEG encoded video over an ATM connection. It was developed at Washington University [2], and is now commercially distributed by STS Technologies, Inc., Bridgeton, MO.

The MMX can transmit and receive video and audio full duplex over an ATM network. The video resolution is 640x480 pixel at a frame rate of 30 fps. For compression and decompression there are two independent chips that implement the M-JPEG algorithm.

The video compression quality factor (Q-factor) is programmable, and with it the bandwidth of the video stream changes.

The built in ATM network interface uses permanent virtual circuits (PVC). The data is sent in raw ATM cells. This method is defined as ATM Abstraction Layer 0 (AAL0). There is no higher level protocol information.

4.4 Modifications for this Project

Two major changes to the MARS were required to replace the MMX. First, the ATM protocol had to be changed from AAL0 to AAL5 because no ATM network card could be found that supported AAL0 under Microsoft Windows. Second, the way video is recorded had to be modified. The video codec board encodes the Motion-JPEG video data into an AVI format file. The MARS server needs additional meta-data to playback a video data file. This meta-data has to be created before MARS can use the AVI file.

4.4.1 AAL5 and Higher Layer Protocols

As mentioned above, the MMX uses the ATM Adaptation Layer 0 to send the video data. Since there is no higher level information sent with AAL0, it is impossible to detect if ATM cells were lost in the network. Use of AAL0 also makes it difficult to implement control schemes in ATM switches (e.g., congestion avoidance), because AAL0 does not support the concept of packets. Therefore, AAL0 is usually not supported by ATM card drivers.

AAL5 frames support packet encapsulation and contain a trailer at the end of each packet. The trailer contains information about the size of the packet. If cells are lost during the transmission, the size of the received packet differs from the size given in the trailer. This allows detection of a corrupted packet. The maximum payload size of an AAL5 packet is 64 kilobytes. More information about ATM Adaptation Layers can be found in [7].

The Windows 95 driver for the Efficient Networks ATM card does not allow direct access to the AAL5 level. The driver for AAL5 with no higher protocols is being developed, but it is currently not available. The easiest alternative way to access the ATM network is

through a User Datagram Protocol (UDP) socket. A UDP-socket is a datagram oriented connection. The reliable stream-oriented Transport Control Protocol (TCP) socket is not useful in this application. If data is lost, then there is no point in retransmitting it, because by the time the loss is detected and a retransmission command is issued the video has advanced so far that the data is of no use anymore.

The Efficient Networks ATM card under NetBSD supports AAL0 and AAL5, but no higher level protocols. An additional protocol is required to communicate with an application in the Microsoft Windows environment. Therefore the server software must simulate this additional protocol stack. The current Windows 95 driver for the Efficient Networks ATM card supports Multiprotocol Encapsulation, Classical IP over ATM or LAN Emulation. The protocol that was found easiest to simulate in the server software is Multiprotocol Encapsulation or Bridged Ethernet according to RFC 1483 section 4.2 [8]. This protocol is used to forward Ethernet packets over an ATM network. The structure of an AAL5 frame for Bridged Ethernet is shown in Figure 5.

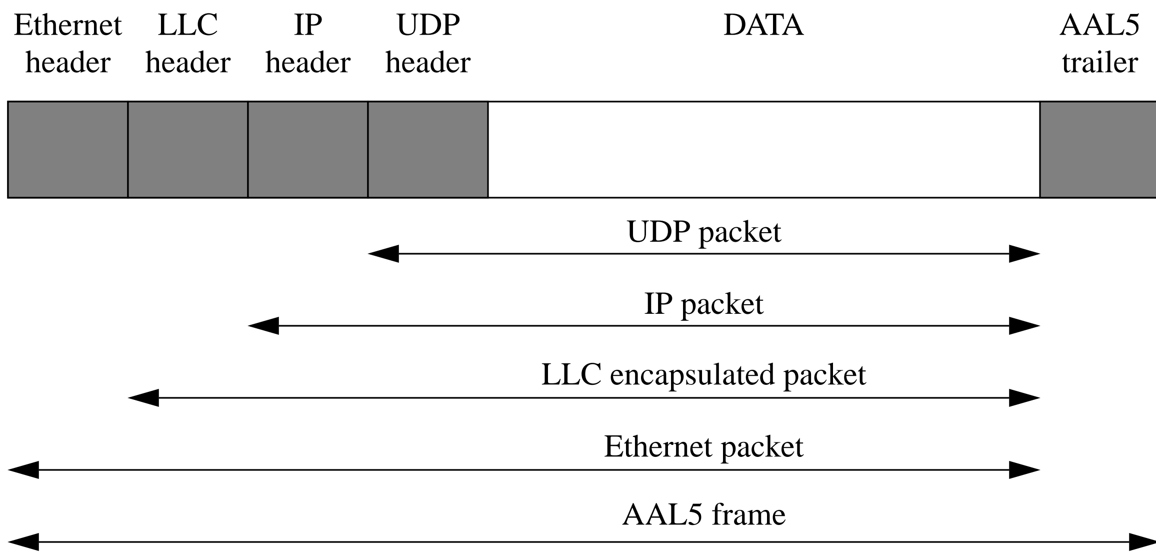

FIGURE 5. Data encapsulation for Bridged Ethernet

The data is prepended with a User Datagram Protocol (UDP) header. It contains the port number used by the receiving socket. This allows the operating system to forward the data to the right socket. The Internet Protocol (IP) header contains information about the internet address of the destination machine. The Logical Link Control (LLC) header consists

of two octets that identify the packet as an IP packet. The Ethernet header contains the physical address of the ATM card (the ATM card is identified by an Ethernet-style address when using RFC 1483).

Although there are several checksums and CRCs in these protocols, only the IP-header requires a correct checksum. All others can be omitted to speed data transfer by eliminating processing overhead.

Since the data transport mechanism now contains Ethernet packets, the maximum size for a packet payload shrinks to 1500 octets (including LLC, IP and UDP header).

4.4.2 Video Recording

In the standard MARS system the video is recorded by the MMX and sent across the ATM network to be stored at the server.

In this modified version the video is first recorded locally with a video capture card which interfaces to the PCI bus of the recording workstation. The video capture card allows the video quality to be reduced to achieve a desired bandwidth. There are also the options to record only one field or only with half resolution (320 x 240 pixels).

The recorded AVI file can then be edited with the commercial software that comes with the capture card. For example the video can be cut to the desired length and a title frame can be added at the beginning.

The AVI file is then parsed to extract information that is required by the server to locate frame boundaries. The detailed process is described in Chapter 5.4.

Finally the video data is sent to the server where it is stored for playback. This can be done using `ftp`. The video and meta data must be stored in the directory `/MARS/accs/username/videoname`. Then it can be accessed by the playback software.

5 Video Acquisition

5.1 Video Capture Hardware

In order to be inexpensive, the video capture card that is used for video recording and playback had to be a commercial off-the-shelf product. Several video codec boards were screened subject to the following criteria:

- PCI-bus
- NTSC video input and output (composite signal, S-Video optional)
- standard Motion-JPEG encoding and decoding
- capture rate at least 30 fps
- capture resolution at least 640x480 pixel
- variable compression ratios
- 4:2:2 color sampling
- drivers for Windows 95 / NT
- price under $1000

Capture cards that fulfill these requirements are: miroVideo DC20 / DC20+, miroVideo DC30 (Miro Computer Products), Bravado 1000 (Truevision), AV Master (Fast Electronic). All of these adapters use the JPEG Image Compression Processor ZR36050 by Zoran Corporation. Therefore the encoded data can be expected to be very similar. For this project the miroVideo DC20 was chosen because it is the least expensive card (around $550) and drivers for Windows 95 and Windows NT are available. It turned out that the DC20 has been discontinued, instead the newer DC20+ is distributed. Unfortunately only the Windows 95 driver for the DC20+ is available at this point. This caused the client platform to be Windows 95 based.

5.2 AVI File Format

In order to be interchangeable, the video data should be stored in a format that is well defined and widely used. A common format in the Microsoft Windows environment is

AVI (Audio Video Interleaved). Most commercial software supports this format. AVI is based on RIFF (Resource Interchange File Format). RIFF is a file format for multimedia introduced by the Microsoft Corporation. The tagged file structure allows interleaving several media streams. Every stream is split into chunks marked with headers identifying to which stream the chunks belong and their length. The chunks of the different streams are then interleaved. The header of the RIFF file contains chunks with information about the number and the type of the streams. The AVI file format is a special form of an RIFF file, as it only contains audio and video chunks. To simplify the project the audio chunks are not considered any further. A structure of an AVI file is shown in Figure 6.

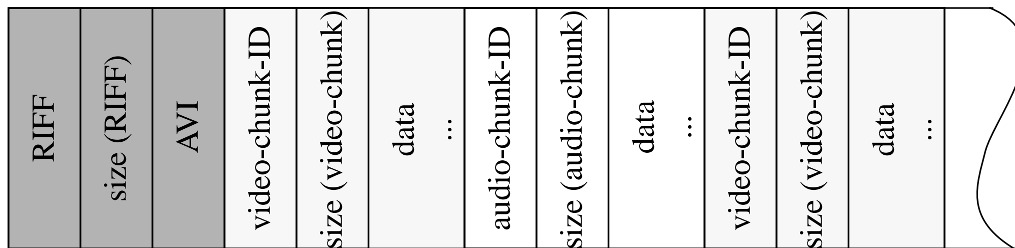

FIGURE 6. Structure of an AVI file

5.3 AVI and M-JPEG

Every frame of a Motion-JPEG encoded video is compressed independently with the JPEG algorithm. Since there is no interframe compression, every frame can be stored independently. Therefore each frame is stored in its own video chunk. To understand the way the JPEG frames are stored, the JPEG algorithm is briefly reviewed.

5.3.1 JPEG Algorithm

The JPEG encoding scheme is a lossy compression algorithm for continuous-tone still images [9]. It takes advantage of characteristics of the human visual system to reduce the amount of data for an image. First, coding is done in the luminance-chrominance-color-space (YCbCr) instead of the usual red-green-blue color space (RGB). The human eye is more sensitive to differences in luminance than differences in chrominance. Therefore the chrominance components are sampled at a lower frequency than the luminance compo-

nent. Second, higher frequency image features are more difficult to compress, but image detail can sometimes be sacrificed in the interest of higher compression rates. Therefore JPEG transforms the image into the frequency domain. There the frequencies are encoded with a precision that depends on their importance for the human visual system.

The following steps describe the complete JPEG encoding algorithm:

1. The image is transformed into the luminance-chrominance color space (YCbCr).
2. The image is split into 8x8 pixel blocks. Each block is encoded independently using the following steps.
3. A discrete cosine transform (DCT) is performed on each color component of the block. The result is an 8x8 matrix for each color component.
4. Every matrix is divided componentwise by a quantization table. The results are rounded to integers. Typically this causes most components to become zero.
5. The rounded matrix is traversed in a zig-zag manner, and the coefficients are listed in a sequence (low frequencies first).
6. The sequence is then encoded with a zero-run-length-code. This shortens the sequence immensely, because most coefficients were zero.
7. Finally a Huffman-code is used to encode this sequence. The Huffman-codes of all components and blocks are concatenated and written to file.

To decode a JPEG image the encoding steps are reversed:

1. Read Huffman-code and decode until a complete sequence is found.
2. Decode zero-run-length-code to sequence of coefficients.
3. Put coefficients back in matrix form.
4. Multiply matrix componentwise with quantization table.
5. Perform a inverse discrete cosine transform (IDCT).
6. Put 8x8 blocks back together.
7. Transform image from YCbCr to RGB color-space.

Luminance and chrominance encoding is done with different quantization tables and Huffman codes. The tables are suggested by the ISO standard, but they can be different for every image. Therefore the tables are included in the header of an JPEG image to allow correct decoding.

Examples and more details for this process can be found in [9] and [10].

5.3.2 AVI Extensions for M-JPEG

The structure of the AVI File Format Extension for M-JPEG [11] is shown in Figure 7.

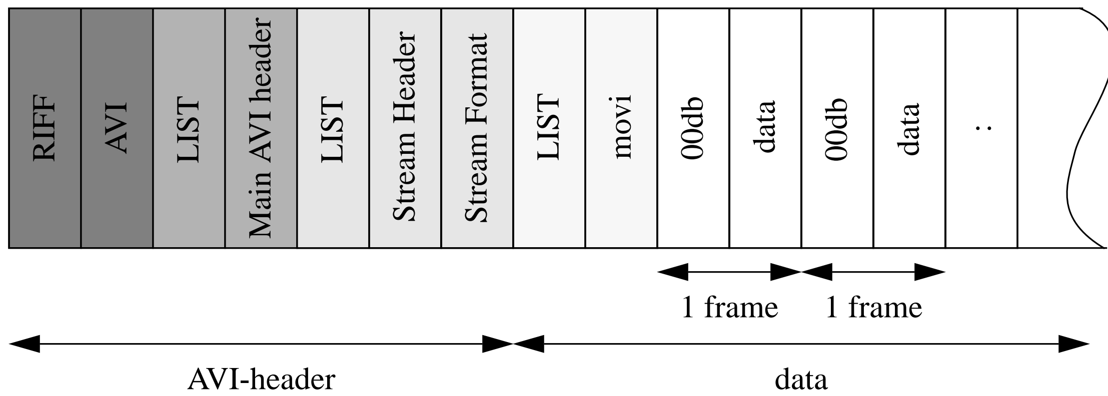

FIGURE 7. AVI file format for M-JPEG

There is one stream of video frames. If there also is an audio stream, then a second Stream Header and Stream Format is required. The identifier for an JPEG compressed frame is "00db". The structure of a single frame is shown in Figure 8.

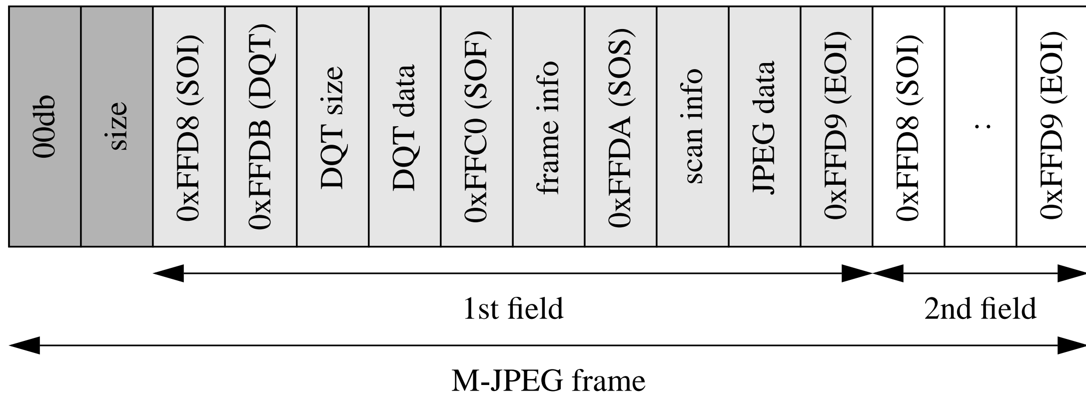

FIGURE 8. Data structure of M-JPEG frame

One frame consists of two fields. A field has the same horizontal resolution as a frame, but only half the vertical resolution. When displayed both fields are interleaved to create the full resolution image. The data structure for the second field is identical to the structure for the first field.

At the beginning of a field there is a Start of Image (SOI) marker. It is followed by the quantization table that was used for the encoding of the particular field. The Define Quantization Table (DQT) marker is followed by the size of the quantization table. Although all quantization tables have the same size, it is possible to have several tables for the different color space components. It is also possible to add the Define Huffman Table (DHT) marker with the Huffman code data after the quantization tables. If this data is not included the decoding application uses its standard Huffman code. Since Huffman tables are several kilobytes in size they are usually not included in the data stream.

The Start of Frame (SOF) marker is followed by information about the resolution of the encoded image, the number of color components and the sampling rate of each component. The Start of Scan (SOS) header contains information about which quantization table is used by which color component. Finally the JPEG data itself is included. The data is concluded by the End of Image (EOI) marker. The markers name is a little misleading, because it is only the end of the first field. The second field has the exact same markers and data as the first field.

5.4 Data Preparation for Server

The MARS server requires the following data and meta-data files in order to be able to send a video stream.

- `.info`: meta-data (duration, number of video, size, ...)
- `.mmjpeg`: meta-data (name of video, corresponding URL, server, port, ...)
- `.mjpeg`: video data
- `.mjpeg.mdata`: meta-data for every frame (offset of start and end, size of frame, ...)

For a standard AVI file the meta-data files have to be created. It is possible to use the AVI file as the `.mjpeg` data file without modification. The meta-data files are created by parsing the AVI file and extracting information about the video. Examples for the meta-data files and the source code for the parsing program can be found in Appendix D.

6 Video Playback

This section describes the software required on the client workstation. The components are shown in Figure 9.

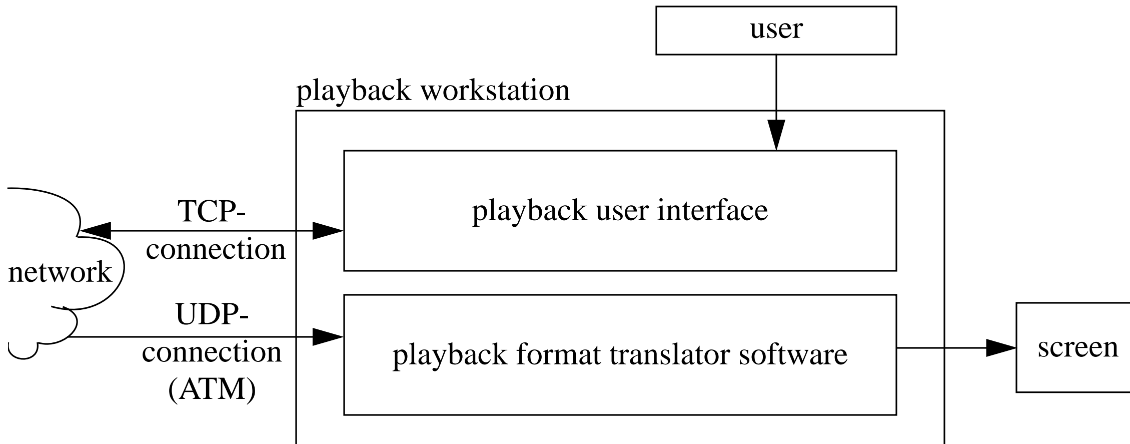

FIGURE 9. Software at client workstation

The playback user interface sends control commands to the video server via a TCP socket. In this project an Ethernet LAN was used for this connection.

The server sends the video data over the ATM network using a datagram-oriented connection. The playback format translator moves the data from the network adapter to the video codec board where it is decoded for displaying.

6.1 ATM under Microsoft Windows

The Efficient Network ATM adapter can be set to operate in one of three different modes under Windows 95 / Windows NT: Multiprotocol Encapsulation, Classical IP over ATM or LAN Emulation. For this project Multiprotocol Encapsulation (according to RFC 1483 [8]) was chosen because it requires few changes in the server software and is easy to use under Windows.

6.1.1 Installation

To install the network adapter open the Network Panel in the Control Panel. Then add a new adapter using the driver disk from the manufacturer. A selection dialog will appear where the ATM client type can be selected. For this project 'RFC1483 PVC' is required. In the Configure menu the Permanent Virtual Circuit (PVC) has to be set to the PVC on which MARS will send the video data. Finally the TCP/IP-protocol has to be added to ATM adapter. The IP address of this network adapter has to be the same IP address that is used in the bridged ethernet header explained in 4.4.1.

6.1.2 ATM Access

Once the ATM network adapter is installed it can be accessed using sockets. To receive the MARS data a socket is setup the following way. First the Internet-socket for a UDP datagram connection has to be created. Then this socket is bound to the IP-address of the ATM adapter. Also a port number has to be specified from which the socket receives data. This port number has to be the same as in the UDP-header that is sent by the MARS server. Finally data packets can be received from this socket. Windows automatically strips off the IP- and UDP-headers. Only the payload data of a packet can be read from a socket.

6.1.3 Frame Transmission

The MARS server sends frame after frame over the ATM connection. A frame has a typical size of several 10kB to a few 100KB, but the Maximum Transmission Unit (MTU) is only 1500 octets. Consequently a frame has to be broken up and transmitted in several packets. It cannot be assured that all packets arrive at the receiving workstation because an unreliable UDP service is used. Therefore a frame protocol is added to every packet. It consists of 6 octets and is described in Figure 10.

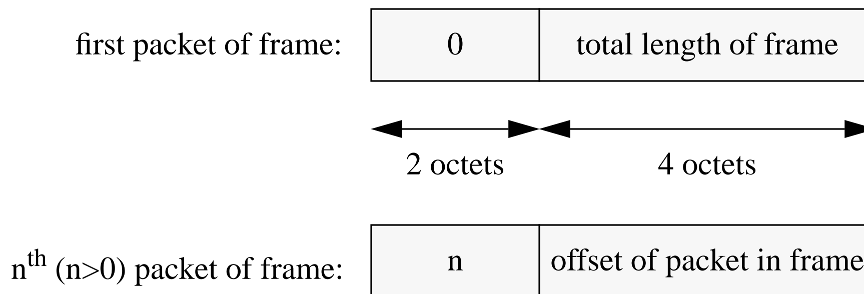

FIGURE 10. Frame protocol

The first field of the protocol header contains the number of the packet. The second field is used for two purposes. When the first packet is transmitted the second field contains the total length of the frame. For all other packets it contains the offset of the data in the frame. That allows easy reassembly of the frame. All numbers are stored in Windows host byte order.

A packet loss can be detected by comparing the total length of the received data with the total expected length transmitted with the first packet. To detect a loss as early as possible the packet numbers can be checked to be continuous. Nevertheless the total size of received data has to be compared to the specified size of the frame, otherwise a possible loss of the last packet cannot be detected.

6.2 Video for Windows

Microsoft Video for Windows supplies a set of Application Program Interfaces (APIs) that allow access to video files and devices. Video for Windows is a standard part of Microsoft Windows 95 and Windows NT.

6.2.1 Media Control Interface

The Media Control Interface (MCI) is a part of Video for Windows. It is used to communicate with multimedia devices that support MCI. MCI commands are high level control commands like Open, Close, Play, Pause, etc. An application that uses MCI does not need to know hardware specific details about the device it wants to use. The device driver is

responsible for translating the MCI command to something that can be understood by the device. The following example opens the file `sample.avi` to be used by the video codec board with the name `avivideo1`:

```
mciSendString("open sample.avi type avivideo1 wait");
```

In this manner an application can be developed without being specifically restricted to a single type of video codec board.

6.2.2 Video Playback over Network

The playback software for the client workstation moves the data from the network to the video codec board. The video board uses the above described MCI device driver. Since MCI device drivers expect to read data from a file, the playback software has to intercept the file I/O of the driver and supply it with the data received from the network.

The I/O interception is possible because all MCI AVI drivers use the `mmioOpen` function to open a file. This function uses different I/O routines depending on the type of file that is being opened. It is possible to install a custom function to be used on files with a certain suffix. The function `mmioInstallIOProc(suffix, function_ptr)` does that. The custom I/O function will be called whenever a file with that suffix (and a '+' after the suffix) is accessed.

During playback, the application sends an MCI command to open a file with the designated suffix. This causes the device driver to use the special I/O function supplied by the playback software. This I/O function reassembles the network data and provides it to the device driver.

6.3 Playback Software

The video playback software consists of two parts. One part is the application that controls the MARS server to initiate a video playback. The other part is the software that receives data from the network and modifies it for the video codec board.

6.3.1 Playback User Interface

This piece of software is a JAVA-applet that was developed in this project. It is stored at the MARS server and can be retrieved like a web-page by a web-browser. It connects to the MARS web-server and starts or stops the sending of the specified video file over the specified PVC. The applet source code can be found in Appendix E.

6.3.2 Playback Format Translator

This piece of software converts frames received from the network to AVI format, suitable for decoding by the JPEG codec. It receives the data packets from the network, checks them and reassembles them to frames. If a frame is incomplete, it is discarded. The complete frames are provided to the MCI device driver as described in 6.2.2.

The MCI device driver must read the AVI header of the video file to start playback. This header can not be transmitted by the MARS server because it does not fit into frame-by-frame transmission scheme used in the server. Therefore, the playback software constructs an AVI header to satisfy the MCI device driver. The source code for the playback software is supplied in Appendix E.

7 Performance

7.1 Overall performance

Initial visual observation of the experimental system indicated a loss of video quality. The performance of the playback system is shown in Figure 11.

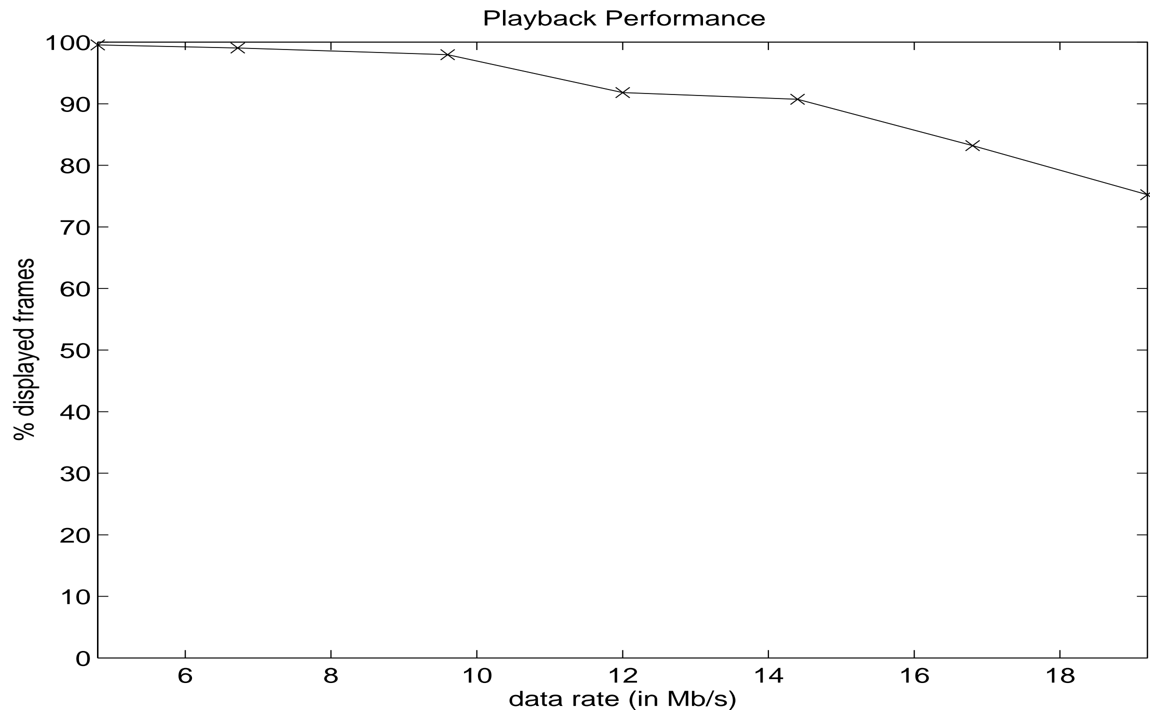

FIGURE 11. Overall performance of playback system

The graph shows the percentage of successfully displayed frames for data rates from 5 to 20 Mb/s. The range of bandwidth corresponds to video quality from low (effects of compression are obvious) to high (effects of compression hardly detectable). A frame is considered successfully displayed when the frame data was passed to the MCI device driver.

For low bandwidths from 5 to 10 Mb/s greater than 97% of the transmitted frames are displayed. The frame loss is not readily detected visually.

For higher data rates the percentage of frame loss increases and the video quality is affected considerably. For a data rate of 20 Mb/s only 75% of the transmitted frames are displayed. The frame loss occurs randomly.

7.2 Performance Analysis

7.2.1 Measurement Outline

To understand the performance of the playback system several measurements were performed. Figure 12 shows a block diagram of the playback workstation. The arrows show the path of the video data through the systems. The dotted lines correspond to places where the video data loss rate was determined.

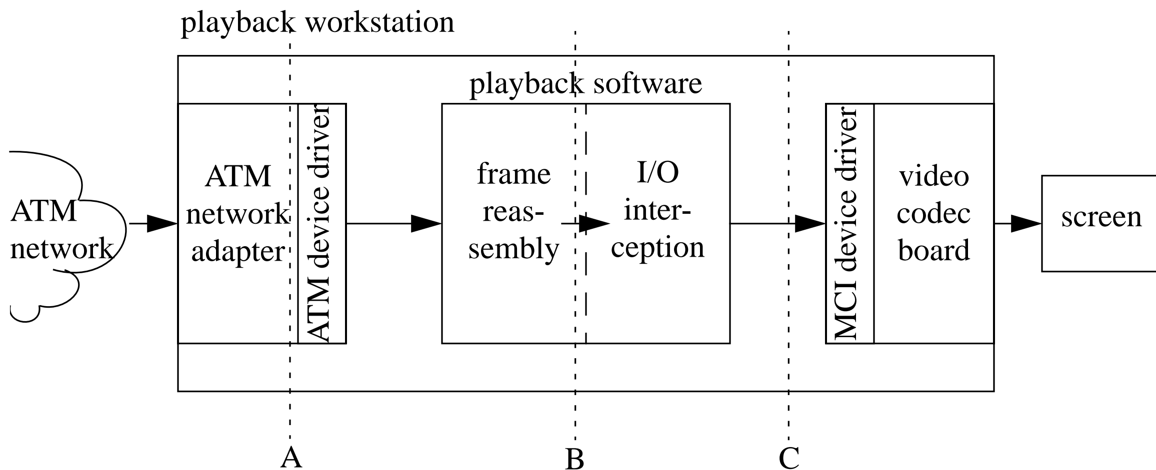

FIGURE 12. Performance measurements in playback system

The measurement of AAL5 packet loss at the ATM network adapter (A) was done with only the monitoring software being active. The playback software and the video codec board were not used to avoid effects caused by these components.

The measurement of video frame loss in the playback software (B) was performed without passing the data to the MCI device driver.

7.2.2 Results

The following results were found:

(A) The ATM network adapter monitoring software was used to determine if data gets lost on arrival at the workstation. This software is supplied by Efficient Networks. It counts the

number of cells that were incorrectly received or dropped. It also monitors the number of complete AAL5 frames.

The tests did not show any loss of cells or frames over a transmission period of 1000 video frames for data rates from 5 to 20 Mb/s.

(B) The playback software was modified to record the number of video frames that were received completely and the number of frames that could not be reassembled because of packet loss. The M-JPEG decoder was disabled for this test to avoid CPU load associated with sending the video data to the codec.

When the playback software only reassembled the frames without passing them on to the MCI device driver, there were no video frame losses recorded over 1000 transmitted frames with data rates from 5 to 20 Mb/s.

(C) The video frame loss rate measured at the MCI device driver interface corresponds to the overall performance shown in Figure 11.

It was observed that frame loss occurs because single packets or very short bursts of packets are lost. This makes it impossible to reassemble the frame. The packet loss is caused by the high system load that is imposed by the MCI device driver. The load causes the operating system to neglect incoming packets from the ATM connection. The analysis of the system load gives a more detailed explanation.

7.2.3 CPU Load during Playback

The CPU load was recorded during the measurements at the three points in the playback system. The results are shown in Figure 13.

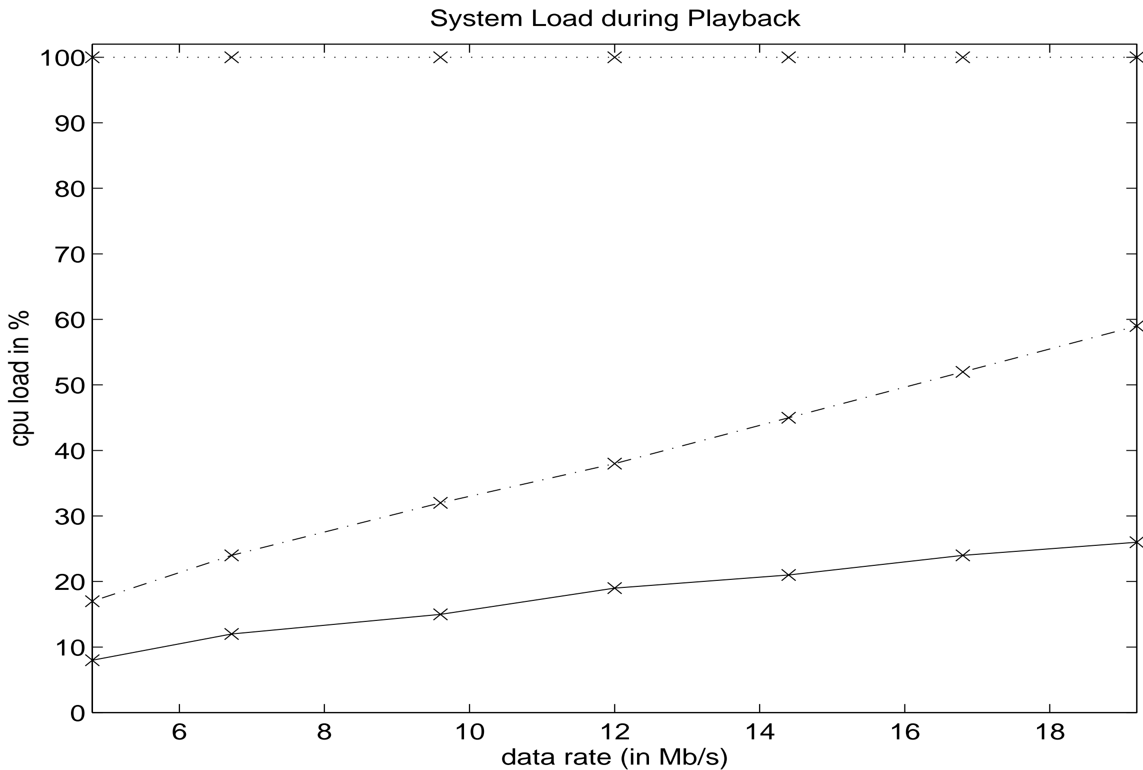

FIGURE 13. CPU Load at playback workstation (A solid, B dot-dashed, C dotted)

When the MCI device driver is not active (A and B) the CPU load is always below 100%. The fact that no packet gets lost in these cases shows that the system has the capacity to handle all incoming packets from the network.

While using the MCI device driver (C) the system load is always 100%. The additional load is induced by the MCI device driver because the playback software processes the data in the same way as in B. In particular there is no “busy-waiting” for actions of the device driver. When the CPU load is 100% the system is too busy to process incoming packets. This would lead to packet loss, and consequently frame loss.

For higher data rates the system can be assumed to be busy with a higher probability. Therefore the frame loss probability increases with the bandwidth of the video data stream. This could explain the decreasing playback quality observed in Figure 11.

8 Summary and Future Work

8.1 Summary

This project investigated acquisition, storage and presentation of digital medical video. An inexpensive off-the-shelf JPEG video codec board and an ATM network interface card in the Microsoft Windows environment is considered as an alternative for the MMX to display stored video from the MARS. The MARS server was modified to distribute a stream of AAL5-encapsulated JPEG video suitable for decoding on a widely available video codec. Multimedia standards like Video for Windows and AVI files were used to achieve a common interface between hardware components. Video playback is achieved using these lower cost hardware components.

Microsoft Windows does not support real-time services. Therefore, the server transmits video frames in periodic intervals to provide the client with a synchronization signal. As long as network delays remain constant, and the client system has sufficient processor and memory bandwidth the video playback is jitter-free.

8.2 Future Work

8.2.1 Playback Performance

To decrease the frame loss rate to a point where it does not visually affect the playback quality, the system load on the client workstation must be lowered. One possibility is to decrease the number of times the video data is copied in the playback workstation. In the current system the data is copied from the ATM network adapter to the system memory. After processing the network protocol header the data is copied to the memory used by the playback software. Then the data is copied to the memory of the MCI device driver. Finally the data is copied onto the video codec board. Ideally, the data should be copied only once from the network adapter to the video codec board. To achieve this, an adjustment of the driver software for the ATM card and the video codec board would be necessary.

Another task of the operating system is the processing of the incoming packets. The header for Ethernet, LLC, IP and UDP have to be processed. This overhead corresponds to the CPU load plotted as A in Figure 13. This might be reduced significantly by using a null-encapsulated AAL5 driver for the ATM network card. Null-encapsulated AAL5 only requires the processing of the AAL5 trailer and no further protocols.

The synchronization of the playback depends on constant network delays. To achieve jitter-free playback over a network with varying delays, the Windows operating system on the client system has to be extended to support real-time services, so that a local clock can be used to drive the repetitive events.

8.2.2 User Interfaces

The graphical user interfaces that were developed for the MARS system use Perl and the Tk module. Perl/Tk is not available for Microsoft Windows. Therefore, a set of GUIs for the playback control and an automatic recording process should be developed.

8.2.3 Higher Resolution

After the playback performance has been improved, a higher resolution video codec adapter can be used to achieve a higher quality video that is required by some medical application. As long as there is a MCI driver available for the video codec board, the developed software can be used.

9 Appendix A: Hardware Specification

The following hardware was used for the MARS server:

- System: Gateway 2000 GP6-200
- Operating System: NetBSD 1.2.1 with extensions described in 4.1.1
- Processor: Intel Pentium Pro (200 MHz)
- Memory: 64 MB RAM
- Video card: MATROX MGA Millenium 2064W
- SCSI-adapter: Adaptec AHA-3940
- Hard drives: Seagate Barracuda 4 (ST15150W, 4GB), Seagate Barracuda 4LP (ST34371W, 4GB)
- Network cards: 3Com 3C905 Ethernet card, Efficient Networks ATM card ENI-155p (driver: `en` version 1.9)

The following hardware was used for the playback workstation:

- System: Gateway 2000 GP6-200
- Operating System: Microsoft Windows 95
- Processor: Intel Pentium Pro (200 MHz)
- Memory: 32 MB RAM
- Video card: MATROX MGA 4MB WRAM
- Hard drive: 1 GB IDE-drive
- Network cards: 3Com 3C905 EtherLink III ISA, Efficient Networks ATM card ENI-155p (driver: RFC1483 version 3.0.1)
- Video codec board: miroVideo DC20 plus (driver version 4.1.4)

10 Appendix B: NetBSD Installation

This part describes the installation of NetBSD on the described hardware and the installation of all software that is required for the MARS server.

10.1 Installing NetBSD release

This part describes the installation of NetBSD release 1.2.1 via FTP. NetBSD is publicly available. The kernel and installation disk have to be created before installation.

First the disk drives have to be partitioned. The disk partitioning program that was the most useful and easily available is `fdisk` for LINUX. Therefore a bootdisk and a rootdisk for the LINUX operating system have to be created. The disks can be downloaded with any LINUX distribution. A URL to a LINUX distribution can be found in Appendix F. After booting LINUX with the bootdisk and rootdisk, the following steps have to be taken:

- start `fdisk`: `fdisk /dev/sda`
- create BSD partition and set system's id to `0xa5` (BSD/386)
- choose option 'edit BSD disklabel'
- compare the drive data that was recognized by `fdisk` with the data that is specified in the manual, if required, edit drive data (it is not unusual for `fdisk` not to get all drive data automatically, for example the spindle speed)
- create three BSD partitions: root, swap and the normal user partitions (recommended sizes are: root 16MB, swap twice memory of system, user rest). Set the appropriate file system type (4.2BSD or higher for root and user partition, swap for swap partition).
- view disklabel, it should look like this:

  ```
   #      size  offset   fstype
  a:     32130      63   4.2BSD  ... (root)
  b:    257040   32193     swap  ... (swap)
  c:   8184647  289233   4.2BSD  ... (remaining space)
  ```

  The sizes are given in sectors (usually 1 sector = 512 bytes). This example is for a 4GB disk drive with 5167 cylinder, 10 heads and 164 sectors/track (= 8473880 sectors total).

The first 63 sectors are used by the disk drive and cannot be used for any partition. The exact sizes and offsets depend on the disk type and the sizes that were chosen for the partitions.

- when everything is correct, write disklabel to disk and exit LINUX.

Now, boot the system with the NetBSD 1.2.1 kernel floppy and installation floppy. The kernel floppy has to be chosen depending on the hardware used in the system. The ‘generic kernel image for Adaptec SCSI-controllers’ was used here.

Next the NetBSD system has to be installed. The installation program that comes with the installation floppy has shown to fail in some cases, therefore the ‘manual’ installation is described here.

The system disk must be formatted and mounted:

- make disk bootable: `disklabel -B /dev/sd0a`
- run `newfs /dev/rsd0a` and `newfs /dev/rsd0c` to find out what parameters are required for block size, i-node size and frag size (here the block size is 16384, i-node size 5957 and frag size 4096)
- format root partition: `newfs -b 16384 -i 5957 -f 4096 /dev/rsd0a`
- mount root partition: `mount -v /dev/sd0a /mnt`
- format user partition: `newfs -b 16384 -i 5957 -f 4096 /dev/rsd0c`
- create user directory: `mkdir -p /mnt/usr`
- mount user partition: `mount -v /dev/sd0c /mnt/usr`

The disks have to be populated with system files:

- extract files:
  `tar --one-file-system -cf - . | (cd /mnt ; tar --unlink -xpf - )`
- create profile-file: `cp /tmp/.hdprofile /mnt/.profile`

The disk drive configuration has to be stored in the fstab-file:

- create fstab for root partition:

  `echo /dev/sd0a / ffs rw 1 1 | sed -e s,//,/, > /mnt/etc/fstab`

- create fstab for user partition:

  `echo /dev/sd0c /usr ffs rw 1 2 | sed -e s,//,/, >> /mnt/etc/fstab`

- `sync`

Now the NetBSD packages can be installed via FTP:

- create temporary directory for packages: `Set_tmp_dir`
- configure Ethernet card with IP-address of system: `ifconfig ep1 128.252.175.62`
- create default route to gateway: `route add default 128.252.175.254`
- ftp to ftp-server that has the NetBSD packages: `ftp 128.252.135.4`
- download `base121`, `comp121`, `etc121`, `man121`, `misc121` and `text121` from the NetBSD release 1.2.1 directory.

Then extract all packages:

- `Extract base121`
- `Extract comp121`
- ...

Now configure system and reboot:

- `Configure`
- `reboot` (reboot with kernel floppy only, press enter when asked for file system floppy)

Finally install kernel to hard disk:

- `copy_kernel`
- `reboot` (without any floppy)

In case some optional or updated packages are downloaded later, install them as follows:

- `cat pkgname* | gunzip | tar -xvf - -C /`

10.2 Compiling NetBSD Kernel

The kernel has to be recompiled in order to adjust the system for different hardware devices and adapters. The hardware configuration is stored in a configuration file. When compiling the kernel all device drivers listed in the configuration file are included in the new kernel.

To create a new configuration file follow these steps:

- `cd /sys/arch/i386/conf`
- choose a configuration or modify existing (here `GENERICADP`)
- `config GENERICADP`
- `cd /sys/arch/i386/compile/GENERICADP`
- `make depend`
- `make`

Now the new kernel is compiled. It is the file `/sys/arch/i386/compile/GENERICADP/netbsd`. To make it the kernel that is used in the system, copy it into the root directory. But first backup the previously used kernel:

- `cp /netbsd /netbsd.old`
- `cp /sys/arch/i386/compile/GENERICADP/netbsd /`
- `reboot`

10.3 Installing NetBSD current

NetBSD current is the version of NetBSD that is currently being improved. New drivers and features are being implemented and tested. It is not guaranteed that NetBSD current works. In this project NetBSD current is required because the driver for the Efficient Networks ATM card, Real-Time Upcalls and Concatenated Disks are not a part of the 1.2.1 release.

Upgrading NetBSD 1.2.1 to NetBSD current:

- download the system source code (as a `.tar.gz` file) from the NetBSD current directory of the ftp-server
- extract all current files into with `gunzip sys.tar.gz | tar -xvf - -C /`
- `cd /usr/src/usr.sbin/config`
- `make`
- `make install`
- `make cleandir`
- `cd /usr/src/usr.bin/gas`
- `make`
- `make install`
- `make cleandir`

Create a kernel configuration file that contains the Efficient Networks ATM:

- `cd /sys/arch/i386/conf`
- create kernel configuration file (here `GENATM`) based on `GENERICADP`, add or comment out lines
- for Efficient ATM card add: `en* at pci? dev ? function ?`
- `config GENATM`
- `cd /sys/arch/i386/compile/GETATM`
- `make depend`
- `make`

Finally copy kernel to root directory and reboot. Now the ATM card should be recognized at boot time. To assign an IP-address to the ATM card run the following command:

- `ifconfig en0 128.252.180.62 netmask 0xffffff00`

The ATM card does not have to specially configured for the RFC1483 protocol that is used by the ATM adapter under Microsoft Windows. The MARS server software supplies the necessary protocol handling.

If the ATM card is the second network card, then NetBSD automatically makes the computer act as a router. This is not necessarily desirable. Therefore the entry for `routed_flag` in `/etc/netstart` should be: `routed_flag=NO`

11 Appendix C: MARS Installation

The MARS server files are available at `leto.wustl.edu`. They are located in the directories `/MARS` and `/usr/MoD`. The software for the playback server and the record daemon have to be copied into the directory tree `/usr/MoD`. If the files are installed in a different directory tree, then the file paths in all source files have to be adjusted accordingly or soft links have to be created. The following list shows the subdirectories and their contents:

- `GUIS` - graphical user interfaces that are being used by the client (Perl script)
- `MISCPerl` - Perl scripts for setup, ...
- `httpd` - WWW-daemon which lets client play videos
- `include` - NetBSD files for ATM support
- `jpeg5b` - routines for extracting single frames
- `marslib` - common library functions
- `mmxd` - client daemon for communication with MMX
- `netpbm` - routines for extracting single frames
- `recordd` - server daemon for recording

The files that are being remotely accessed have to be located in `/MARS`. This includes the video data as well as the html documents for WWW-server. The subdirectory structure is:

- `accs` - accounts where the video data is stored
- `conf` - some configuration files
- `html` - the html files for the WWW-daemon
- `root` - common images, ...

11.1 Activating MARS

To make the MARS server accessible via a network the WWW-server and record-daemon have to be running. To start the WWW-server execute `/usr/MoD/httpd/httpd`. To start the record-daemon execute `/usr/MoD/recordd/recordd`.

The WWW-daemon is configured to run on port 10000 of the server. When using for example Netscape as a web browser the URL to access it has to be `http://server.domain:10000`, so the browser uses port 10000 instead of the standard port 80. To change the port used, modify the line `Port 10000` in file `/MARS/conf/httpd.conf` accordingly.

The recording service can be accessed via the web server.

11.2 Compiling MARS

The following instructions build the httpd-executable for NetBSD:

- `cd /usr/MoD/httpd/src`
- `make clean`
- `make netbsd_free_shm_win`

The following instructions build the recordd-executable:

- `cd /usr/MoD/recordd`
- `make clean`
- `make free_shm`

11.3 Miscellaneous about MARS

If the MMX is used for recording and playback, Perl and Tk are required for the user interface. Perl and Tk are not required for the system that has been developed in this project. The following instructions describe the installation:

- download the newest Perl sources and extract sources to some directory
- `sh Configure`
- `make`
- `make test`
- `make install`

The Tk module is installed as follows (requires X11 to be installed):

- download newest Tk source and extract sources to some directory
- `perl Makefile.PL`
- `make`
- `make test`
- `make install`

12 Appendix D: Video Meta Data Software

Example for `.info` file:

```
DURATION 5
NOVIDFRAMES 180
SIZE 3723072
FORMAT MJPEG
QFACTOR 0
AUDIO_SAMPLE_RATE 0
```

Example for `.mmjpg` file:

```
IMAGE root/movie2.gif
SERVER leto.wustl.edu
PORT 10000
DURATION 5
NOVIDFRAMES 180
SIZE 3723072
FORMAT MJPEG
QFACTOR 0
AUDIO_SAMPLE_RATE 0
```

parseAVI.c:

```
/* This program parses an AVI file and generated metadata file      */
/* for the MARS server.                                              */
/* Microsoft Visual C++ 4.0                                          */
/* August 1997 by Tilman Wolf                                        */

#include <windows.h>
#include <stdio.h>
#include <winsock.h>

typedef struct {                              /* standard AVI header */
    unsigned long dwMicroSecPerFrame;
    unsigned long dwMaxBytesPerSec;
    unsigned long dwReserved1;
    unsigned long dwFlags;
    unsigned long dwTotalFrames;
    unsigned long dwInitialFrames;
    unsigned long dwStreams;
    unsigned long dwSuggestedBufferSize;
    unsigned long dwWidth;
    unsigned long dwHeight;
    unsigned long dwScale;
    unsigned long dwRate;
```

```
    unsigned long dwStart;
    unsigned long dwLength;
} MainAVIHeader;

typedef struct file_header {                /* header for .mjpeg.mdata file */
  long duration ;                           /* Seconds */
  long no_frames;                           /* number of frames */
  long size;                                /* in bytes */
  char format[32];                          /* MJPEG */
} FileHeader;

typedef struct meta_data {                  /* entry in .mjpeg.mdata for evey frame */
  int no;                                   /* frame number */
  int ev_s;                                 /* start of even field */
  int ev_e;                                 /* end of even field */
  int odd_s;                                /* start of odd field */
  int odd_e;                                /* end of odd field */
  int s;                                    /* start of frame */
  int e;                                    /* end of frame */
  int sz;                                   /* size of frame */
  int loss;                                 /* obsolete */
} MetaData;

int pattern(FILE *in, char *buffer, int size) {
        int i;                              /* tries to find pattern in file */
        char byte;

        for (i=0; i<size; i++) {
                fread(&byte,1,1,in);
                if (byte != buffer[i]) {
                        fseek(in,-(i+1),SEEK_CUR);
                        return 0;
                }
        }
        return 1;
}

void skipJunk(FILE *in) {                   /* skips JUNK chunk in AVI file */
        int size;
        while ((pattern(in,"JUNK",4)) != 0) {
                fread(&size,4,1,in);
                fseek(in,size,SEEK_CUR);
        }
}

int PASCAL WinMain(HANDLE hInst, HANDLE hPrev, LPSTR szCmdLine, int sw) {
        FILE *avi, *mdata, *info, *mmjpg;
        char name[100];
        char file[130];
        unsigned long frames;
        unsigned long riffsize, hlistsize, dlistsize, framesize,
                avihsize;
        MainAVIHeader AVIHeader;
        FileHeader FHeader;
        MetaData MData;

        if (sw != 1) {                      /* checks for one argument */
                printf("argument missing: filename (without extension)\n");
                exit(-1);
        }
        strcpy(name,szCmdLine);

        sprintf(file,"%s.avi",name);        /* open files */
        avi = fopen(file,"rb");
        sprintf(file,"%s.mjpeg.mdata",name);
```

```
mdata = fopen(file,"wb");

pattern(avi,"RIFF",4);                /* start parsing AVI file header */
fread(&riffsize,4,1,avi);
pattern(avi,"AVI ",4);
pattern(avi,"LIST",4);
fread(&hlistsize,4,1,avi);
pattern(avi,"hdrl",4);
pattern(avi,"avih",4);
fread(&avihsize,4,1,avi);
fread(&AVIHeader,sizeof(AVIHeader),1,avi);
fseek(avi,hlistsize-(12+sizeof(AVIHeader)),SEEK_CUR);

FHeader.duration = htonl((AVIHeader.dwTotalFrames *
                         AVIHeader.dwMicroSecPerFrame) / 1000000);
FHeader.no_frames = htonl(AVIHeader.dwTotalFrames);
FHeader.size = htonl(8 + riffsize);
strcpy(FHeader.format,"MJPEG");
fwrite(&FHeader,sizeof(FHeader),1,mdata);

skipJunk(avi);
pattern(avi,"LIST",4);
fread(&dlistsize,4,1,avi);
pattern(avi,"movi",4);                /* start parsing AVI file data */
frames = 0;
while ((pattern(avi,"00db",4)) != 0) {
        fread(&framesize,4,1,avi);

        MData.no = htonl(frames++);
        MData.ev_s = 0;
        MData.ev_e = 0;
        MData.odd_s = 0;
        MData.odd_e = 0;
        MData.s = htonl(ftell(avi) - 8);
        MData.e = htonl(ftell(avi) + framesize);
        MData.sz = htonl(framesize + 8);
        MData.loss = 0;
        fwrite(&MData,sizeof(MData),1,mdata);

        fseek(avi,framesize,SEEK_CUR);
        skipJunk(avi);
}
fclose(mdata);

sprintf(file,"%s.info",name);         /* create .info file */
info = fopen(file,"wb");
fprintf(info,"DURATION %d\10",ntohl(FHeader.duration));
fprintf(info,"NOVIDFRAMES %d\10",ntohl(FHeader.no_frames));
fprintf(info,"SIZE %d\10",ntohl(FHeader.size));
fprintf(info,"FORMAT MJPEG\10");
fprintf(info,"QFACTOR 0\10");
fprintf(info,"AUDIO_SAMPLE_RATE 0\10");
fclose(info);

sprintf(file,"%s.mmjpg",name);        /* create .mmjpg file */
mmjpg = fopen(file,"wb");
fprintf(mmjpg,"IMAGE root/movie2.gif\10");
fprintf(mmjpg,"SERVER leto.wustl.edu\10");
fprintf(mmjpg,"PORT 10000\10");
fprintf(mmjpg,"DURATION %d\10",ntohl(FHeader.duration));
fprintf(mmjpg,"NOVIDFRAMES %d\10",ntohl(FHeader.no_frames));
fprintf(mmjpg,"SIZE %d\10",ntohl(FHeader.size));
fprintf(mmjpg,"FORMAT MJPEG\10");
fprintf(mmjpg,"QFACTOR 0\10");
fprintf(mmjpg,"AUDIO_SAMPLE_RATE 0\10");
```

```
        fclose(mmjpg);
        fclose(avi);

        sprintf(file,"rename %s.avi %s.mjpeg",name,name);
        system(file);                         /* rename .avi to .mjpeg */

        return 0;
}
```

13 Appendix E: Playback Software

13.1 JAVA-Applet to control MARS

HTML-page for applet:

```
<HTML>
<TITLE>Multimedia on Demand Playback</TITLE>
<BODY>
<applet code="MoDPlay.class" width=540 height=270>
</applet>
</BODY>
</HTML>
```

Applet:

```
/* This JAVA-Applet controls the playback of video from */
/* the MARS server.                                     */
/* August 1997 by Tilman Wolf                           */

import java.applet.*;
import java.awt.*;
import java.net.*;
import java.io.*;

public class MoDPlay extends Applet {
  static Socket s;
  static OutputStream out;
  static InputStream in;
  PrintStream printout;
  ByteArrayInputStream inbuf;
  TextField videoname,username,pvcno;
  String name,user;
  int packet,pvc;

  public void init() {                 /* create GUI */
    setLayout(new FlowLayout());
    add(new Label("user:"));
    username = new TextField("tilman",10);
    add(username);
    add(new Label("video:"));
    videoname = new TextField("sample",10);
    add(videoname);
    add(new Label("PVC:"));
    pvcno = new TextField("512",4);
    add(pvcno);
    add(new Button("OPEN"));
    add(new Button("CLOSE"));
    user = "tilman";
    name = "sample";
    packet = 1400;
    pvc = 512;
  }

  public static void main(String argv[]) {
    System.out.println("Please run this JAVA class as an Applet.");
  }

  public boolean handleEvent(Event evt) {
    if (evt.id == Event.WINDOW_DESTROY) System.exit(0);
    return super.handleEvent(evt);
```

```
}

public boolean action(Event evt, Object obj) {
  if (obj.equals("OPEN")) {          /* open connection to */
                                    /* MARS server */
    s = connectServer("leto.wustl.edu",10000);
    try {
      out = s.getOutputStream();
      in = s.getInputStream();
    }
    catch(IOException e) {
      System.err.println("Error: unable to get I/O stream");
      System.exit(-1);
      out = (OutputStream)null;
      in = (InputStream)null;
    }
    printout = new PrintStream(out);
    sendOpen(name,packet,pvc,501);
  }
  else if (obj.equals("CLOSE")) {    /* close connection to */
                                     /* MARS server */
    try {
      s.close();
    }
    catch(IOException e) {
      System.err.println("Error: unable to close socket");
      System.exit(-1);
    }
  }
  else if (evt.target.equals(videoname)) { /* input on GUI */
    name = videoname.getText();
  }
  else if (evt.target.equals(username)) {
    user = username.getText();
  }
  else if (evt.target.equals(pvcno)) {
    pvc = Integer.parseInt(pvcno.getText());
  }
  else return super.action(evt,obj);
  return true;
}


private static Socket connectServer(String servername,int serverport) {
  InetAddress serverIP;            /* creates socket connection */
                                   /* to MARS */
  try {
    serverIP = InetAddress.getByName(servername);
  }
  catch(UnknownHostException e) {
    System.err.println("Error: unknown server "+servername);
    System.exit(-1);
    return (Socket)null;
  }

  try {
    s = new Socket(serverIP,serverport);
  }
  catch(IOException e) {
    System.err.println("Error: could not open socket");
    System.exit(-1);
  }

  return s;
```

```
  }

  private void sendOpen(String video, int size, int vidvci, int audvci) {
    String cmd;                        /* sends command to play video */

    cmd = "GETSTREAM accs/"+user+"/"+video+"/"+video+".mjpeg?VIDEO+"+
          "HARDWARE+IMAGE=/root/movie2.gif+RAWATM+PROTO=AAL0+"+
          "PACKETSIZE="+size+"+0+"+vidvci+"+0+"+
          audvci+"+LOOPPLAY HTTP/1.0\n\n";
    printout.print(cmd);
  }

  private void sendLoop() {
    String cmd;

    cmd = "GETSTREAM COMMAND=LOOPFLAG HTTP/1.0\n\n";
    printout.print(cmd);
  }

  private void sendClose() {
    String cmd;

    cmd = "GETSTREAM COMMAND=CLOSE HTTP/1.0\n\n";
    printout.print(cmd);
  }

}
```

13.2 Playback Format Translator

```
/* This program receives video data from an ATM connection and forwards  */
/* it to the video capture card. The AVI header that is required to play */
/* the video data is loaded (filename is argument in command line) and   */
/* the AVI-index is 'faked'. Depending on the length of the faked index  */
/* it takes longer until the MCI device driver has to seek back. The     */
/* seeking might disrupt video playback, therefore it's better to have a */
/* long index. If the index is too long, the interactive reaction to a   */
/* click on the stop button takes too long.                             */
/* To link the object code be sure to include winsock32.lib, vfw32.lib   */
/* and winmm.lib.                                                       */
/* Microsoft Visual C++                                                 */
/* August 1997 by Tilman Wolf                                           */

#include <windows.h>
#include <time.h>
#include <sys/timeb.h>
#include <stdio.h>
#include <vfw.h>
#include <mmsystem.h>

#define FRAMESIZE 100000

typedef struct Frame_T *Frame;

struct Frame_T {                 /* frame buffer, can be used as ring buffer */
       Frame next;
       int size;
       char *data;
};

char *header, *index;
int headersize, indexsize, idxoffset;
SOCKET s;
Frame F;
```

```
void recvFrame(SOCKET s, Frame help) {        /* function to receive frame from */
    static unsigned short int expno;          /* open network connection */
        static unsigned long int readsize,expoff; /* blocks until frame is received */
        static unsigned short int readno = 1;
        static int success = 0;
        static int r;
        static BYTE buffer[1500];

        while (success == 0) {               /* repeat until complete frame */
                while (readno != 0) {           /* find first packet of frame */
                        r = recv(s,buffer,1500,0);
                        memcpy(&readno,&(buffer[0]),2);
                        memcpy(&readsize,&(buffer[2]),4);
                }
                help->size = readsize; /* store expected size of frame */
                memcpy(&((help->data)[0]),&(buffer[6]),r-6); /* copy data */
                expno = 1;                          /* next expected packet is no 1 */
                expoff = r-6;                       /* next expected offset */
                r = recv(s,buffer,1500,0);
                memcpy(&readno,&(buffer[0]),2);   /* read next packet no */
                memcpy(&readsize,&(buffer[2]),4); /* read next packet offset */
                while ((readno == expno)&&(readsize == expoff)) { /* loop as long as the */
                        memcpy(&((help->data)[expoff]),&(buffer[6]),r-6);/* expected */
                        expno++;                                        /* packets come */
                        expoff = readsize + r-6;
                        r = recv(s,buffer,1500,0);
                        memcpy(&readno,&(buffer[0]),2);
                        memcpy(&readsize,&(buffer[2]),4);
                }
                if (expoff != (unsigned int)help->size) { /* data is missing */
                        success = 0;                    /* discard frame */
                } else {                               /* complete frame */
                        success = 1;                    /* accept frame */
                }
        }
        success = 0;
}

Frame allocFrames(int n) {          /* allocate frame memory */
        int i;
        Frame ring,help,start;

        start = (Frame)malloc(sizeof(struct Frame_T));
        start->size = -1;
        start->data = (char *)malloc(FRAMESIZE);
        help = start;
        for (i=1;i<n;i++) {
                ring = (Frame)malloc(sizeof(struct Frame_T));
                ring->size = -1;
                ring->data = (char *)malloc(FRAMESIZE);
                help->next = ring;
                help = ring;
        }
        help->next = start;
        return start;
}


void PlayAVI(HANDLE hInst){          /* initiates playback on MCI device */
        HWND hand;                      /* creates also STOP button */
        MSG msg;
        char err[1000],ret[1000],com[1000];
        int res,playing;

        hand = CreateWindow(
```

```
                "BUTTON",
                "STOP",
                WS_OVERLAPPEDWINDOW,
                800, 500,
                200, 200,
                (HWND)NULL,
                (HMENU)NULL,
                hInst,
                (LPSTR)NULL,
                );
        if (hand == NULL) return;
        ShowWindow(hand, SW_SHOW);

        wsprintf(com, "open VIDEO.MRS+ type avivideo1 alias avi wait");
        res = mciSendString(com,ret,1000,(HANDLE)NULL); /* open MCI device */
        if (res != 0) {
                mciGetErrorString(res,err,1000);
                wsprintf(com, "close avi wait");
                res = mciSendString(com,ret,1000,(HANDLE)NULL);
                return;
        }

        playing = 1;
        while (playing) {
                wsprintf(com, "play avi wait");            /* play as long as fake index */
                res = mciSendString(com,ret,1000,(HANDLE)NULL);
                if (res != 0) {
                        mciGetErrorString(res,err,1000);
                        wsprintf(com, "close avi wait");
                        res = mciSendString(com,ret,1000,(HANDLE)NULL);
                        return;
                }

                wsprintf(com, "seek avi to start wait"); /* seek beginning of pseudo file */
                res = mciSendString(com,ret,1000,(HANDLE)NULL);
                if (res != 0) {
                        mciGetErrorString(res,err,1000);
                        wsprintf(com, "close avi wait");
                        res = mciSendString(com,ret,1000,(HANDLE)NULL);
                        return;
                }

                if (PeekMessage(&msg, hand, 0, 0, PM_REMOVE)) {/*check if STOP was pressed*/
                        if ((msg.message == WM_QUIT)||(msg.message == WM_RBUTTONDOWN)) {
                                playing = 0;
                        }
                        if (msg.message == WM_LBUTTONDOWN) {
                                playing = 0;
                        }
                        TranslateMessage(&msg);
                        DispatchMessage(&msg);
                }
        }

        wsprintf(com, "close avi wait");              /* close MCI device */
        res = mciSendString(com,ret,1000,(HANDLE)NULL);
        if (res != 0) {
                mciGetErrorString(res,err,1000);
                return;
        }
}

char *getAVIHeader(char *filename, int *size) {    /* reads AVI header file */
        FILE *hfile;
        int i;
```

```
        unsigned char byte;
        char *header;

        hfile = fopen(filename,"rb");
        i=0;
        while (!feof(hfile)) {
                fread(&byte,1,1,hfile);
                i++;
        }
        i--;
        *size = i;
        header = (char *)malloc(i);
        fseek(hfile,0,SEEK_SET);
        fread(header,i,1,hfile);
        return header;
}

char *getPseudoIndex(int entries, int size, int *idxsize) {
        int i,*help;                       /* creates pseudo index and adjusts header */
                                           /* entries = # of frames in index, size =  */
                                       /* max size of one frame */
        help = (int *)malloc(entries*4*4+8);
        help[0] = 0x31786469;              /* idx1 */
        help[1] = entries*4*4;
        for (i=0; i<entries; i++) {
                help[(4*i)+2] = 0x62643030;    /* 00db */
                help[(4*i)+3] = 0x00000010;    /* keyframe */
                help[(4*i)+4] = size*i + headersize;
                help[(4*i)+5] = size;
        }
        idxoffset = headersize + size*entries;
        *idxsize = (entries*4*4+8);
        ((int *)header)[1]=idxoffset+*idxsize-8; /* length of RIFF */
        ((int *)header)[(headersize/4)-2]=idxoffset-(headersize-4);/* length of data LIST*/
        ((int *)header)[12]=entries;        /* number of frames in AVI header */
        ((int *)header)[35]=entries;        /* number of frames in stream header */

        return (char *)help;
}

static char *lpData;
static long filesize;
int fake[2];
int getIndex;

LRESULT CALLBACK IOProc(LPMMIOINFO lpMMIOInfo, UINT uMessage,
                                            LPARAM lParam1, LPARAM lParam2) {
                                         /* intercepted file I/O of MCI device */
        static byte *buffer;

        switch (uMessage) {                     /* uMessage contains I/O command */
        case MMIOM_OPEN:
                lpMMIOInfo->lDiskOffset = 0;
                return 0;

        case MMIOM_CLOSE:
                return 0;

        case MMIOM_READ:
                if (lpMMIOInfo->lDiskOffset >= (idxoffset + indexsize)) { /* eof */
                        return 0;
                } else if (lpMMIOInfo->lDiskOffset < headersize) {  /* read header */
                        memcpy((void *)lParam1,
                                &(header[lpMMIOInfo->lDiskOffset]), lParam2);
                } else if (lpMMIOInfo->lDiskOffset >= idxoffset) {    /* read index */
```

```
                    memcpy((void *)lParam1,
                            &(index[lpMMIOInfo->lDiskOffset - idxoffset]), lParam2);
            } else {                                            /* read data */
                    if (lParam2 == 8) {                         /* frame header */
                            recvFrame(s,F);                 /* read new frame */
                            memcpy((void*)lParam1, F->data, lParam2);
                    } else {                                    /* frame data */
                            if (lParam2 + 8 > F->size) return -1;
                            memcpy((void *)lParam1, (F->data)+8, lParam2);
                    }
            }
            lpMMIOInfo->lDiskOffset += lParam2;
            return lParam2;

    case MMIOM_SEEK:
            switch (lParam2) {
            case SEEK_SET:
                    lpMMIOInfo->lDiskOffset = lParam1;
                    break;

            case SEEK_CUR:
                    lpMMIOInfo->lDiskOffset += lParam1;
                    break;

            case SEEK_END:
                    lpMMIOInfo->lDiskOffset = idxoffset + indexsize - 1 - lParam1;
                    break;
            }
            return lpMMIOInfo->lDiskOffset;

    default:
            return -1;
    }
}


void SetupIOProc() {                /* setups function to intercept MCI I/O */
    LPMMIOPROC res;                 /* for files with extension 'MRS+' */
    AVIFileInit();
    res = mmioInstallIOProc(mmioFOURCC('M','R','S',' '), (LPMMIOPROC) IOProc,
            MMIO_INSTALLPROC | MMIO_GLOBALPROC);
}

void CleanIOProc() {                /* stop intercepting I/O */
    mmioInstallIOProc(mmioFOURCC('M','R','S',' '), NULL, MMIO_REMOVEPROC);
    AVIFileExit();
}

SOCKET SetupSocket() {              /* setup ATM network connection */
    SOCKET s;
    SOCKADDR_IN sin;
    WSADATA wsaData;
    int r,size;

    WSAStartup(MAKEWORD(1, 1),&wsaData);
    s = socket(PF_INET, SOCK_DGRAM, 0);
    if (s < 0)
            return 0xffffffff;

    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = htonl(0x80fcb43f); /* IP-Address of ATM card */
    sin.sin_port = htons(4242);             /* port number */

    r = bind(s, (struct sockaddr *)&sin, sizeof(sin));
    if (r < 0)
```

```
                return 0xffffffff;

        size = FRAMESIZE;                        /* set receive buffer size */
        r = setsockopt(s,SOL_SOCKET,SO_RCVBUF,(char *)&size,sizeof(size));
        return s;
}

void CloseSocket(SOCKET s) {                    /* close network connection */
        closesocket(s);
        WSACleanup();
}

int PASCAL WinMain(HANDLE hInst, HANDLE hPrev, LPSTR szCmdLine, int sw) {
        char headerfile[100];

        if (sw != 1) {                           /* checks for one argument */
                printf("one argument expected: AVI header file name\n");
                return -1;
        }
        strcpy(headerfile,szCmdLine);

        F = allocFrames(1);             /* allocate frame buffer */
        s = SetupSocket();              /* open network connection */
        SetupIOProc();                  /* intercept MCI I/O */

        header = getAVIHeader(headerfile,&headersize);    /* read AVI header */
        index = getPseudoIndex(500,FRAMESIZE,&indexsize); /* create pseudo-index */

        PlayAVI(hInst);                 /* play video */

        CleanIOProc();                  /* stop intercepting */
        CloseSocket(s);                 /* close network connection */

        return 0;
}
```

14 Appendix F: Useful URLs

ATM:

- Efficient Networks:

  http://www.efficient.com

- RFC1483:

  http://ds.internic.net/rfc/rfc1483.txt

- ATM under NetBSD:

  http://www.ccrc.wustl.edu/pub/chuck/bsdatm/wucs.html

AVI:

- AVI file structure:

  http://www.intech.it/people/lapus/avi.html

- AVI file format extension for M-JPEG:

  http://www.matrox.com/videoweb/odml102.html

- How to use MCI to play AVI files from memory (Microsoft Article Q155360):

  http://www.microsoft.com

NetBSD:

- NetBSD home page:

  http://www.netbsd.org

Video for Windows:

- miroVIDEO DC20 SDK:

  http://www.miro.com/e/e3-drivers/e33-digitalvideo/dc20sdke.html

15 Bibliography

[1] Buddhikot, Milind M., Parulkar, Guru M., Kumar, Srihari S., Rangan, P. Venkat: "Design of Large Scale Multimedia-On-Demand Storage Servers and Storage Hierarchies", Handbook of Multimedia Information Management, Prentice Hall, 1997.

[2] Richard, William D., Cox, Jerome R. Jr., Engebretson, A. Maynard, Fritts, Jason, Gottlieb, Brian L., Horn, Craig: "The Washington University MultiMedia eXplorer", Department of Computer Science at Washington University, August 1993.

[3] Blaine, G. James, Dodge, John H., Beardslee, Brian, Goodgold, Ken: "Project Spectrum - Technology Alliance for the Emerging Integrated Health System", Healthcare Information and Management Systems Society, Chicago, 1996

[4] Gohel, Nilesh R.: "Medical Video Transmission via Bandwidth Constrained ATM Networks", Sever Institute of Technology at Washington University, May 1994.

[5] Cranor, Charles D.: "Integrating ATM Networking into a BSD Operating System", Department of Computer Science at Washington University in St. Louis, July 1993.

[6] Chen, X. Jane: "Enhancements to 4.4 BSD UNIX for Efficient Networked Multimedia in Project MARS", Department of Computer Science at Washington University, July 1997.

[7] de Prycker, Martin: "Asynchronous Transfer Mode - Solution for Broadband ISDN", Prentice Hall International Ltd., 1995.

[8] Heinanen, Juha: "Multiprotocol Encapsulation over ATM Adaptation Layer 5", Request for Comments: 1483, July 1993.

[9] International Standard DIS 10918-1: "Digital Compression and Coding of Continuous-Tone Still Images - Part I: Requirements and Guidelines", International Organization for Standardization, 1994.

[10] Pennebaker, William B., Mitchell, Joan L.: "JPEG - Still Image Data Compression Standard", Van Nostrand Reinhold, New York, 1992.

[11] OpenDML AVI M-JPEG File Format Subcommitee, "OpenDML AVI File Format Extensions - Version 1.02", February 1996.