Berkeley Multimedia Research Center
TR 161, August, 2001

Production Quality Internet Television

Matthew R. Delco
Computer Science Division
University of California, Berkeley
delco@cs.berkeley.edu

Abstract

This report describes an inexpensive system for transmitting production quality television over IP-based networks using RTP. The quality of both the audio and video is indistinguishable from conventional broadcast television, although the system does allow for various quality levels of audio and video. A detailed explanation is given of the techniques used by the system to maintain synchronization of audio and video over extended periods of time, and to gracefully handle (and conceal) the loss of data within the network.

1 Introduction

Computer networks have become an essential aspect of computing, and they are being continually improved and enhanced to support an increasing bandwidth capacity. This increase in capacity allows new types of services to be deployed. During the past decade there has been an increasing focus on using computer networks for audio and video communication. This trend can be attributed to increasing network bandwidth (e.g., Internet2) and faster processors, as well as the deployment of an efficient packet distribution scheme called *IP Multicast* [Deering and Cheriton, 1990] [Deering et. al., 1996].

Most network-based audio and video tools, such as *vic* [McCanne and Jacobson, 1995] and *vat* [Jacobson and McCanne, 1992], were designed to operate within the bandwidth constraints of the networks in use at the time these applications were originally developed. They focus on using high compression and low bandwidth transmissions. For example, *vat* uses a 64Kbps stream for transmitting telephone-quality audio, and *vic* commonly uses 200-500Kbps for transmitting a video image that is one-quarter the size of a television image and refreshed a few times per second rather than 30 times per second.

Given today's computer networks, it is reasonable to question whether it is now possible to transmit "production quality" television across these networks. For the purposes of this report, the phrase *production quality* refers to audio and video that is indistinguishable from conventional NTSC television. More specifically, the goal is to transmit full-sized images (720x480) as interlaced video at the conventional 59.94 fields per second. With the advent of Internet2 and Gigabit Ethernet, it is likely that bandwidth constraints will not be a limiting factor in attaining this goal. However, other issues such as packet loss, media synchronization, and the limitations of OS drivers and APIs, bring a new (and unexpected) level of complexity to the prospect of transmitting and receiving production quality television.

This report describes a new system, called *RTP Television* (RTPtv), for transmitting synchronized audio and video RTP streams [Schulzrinne et al., 1996] across IP-based networks [Cerf and Kahn, 1974] [ISI, 1981]. RTPtv is ideal for transmitting television over long distances and is appropriate for use within a large television production environment. In addition, RTPtv can also be used for interactive video conferences that require minimal end-to-end latency. RTPtv transmits television using RTP payload formats for *Motion JPEG* (MJPEG) [Berc et al., 1998] video and uncompressed *Linear 16* (L16)

audio [Schulzrinne, 1996] (for reasons that will be described later). A typical video stream uses 5-10Mbps for CIF video and 10-20Mbps for D1 video, while a typical stereo audio stream is 625Kbps (375Kbps for monaural audio).

The RTPtv streams can be transmitted via multicast and/or unicast. Although multicast is the ideal carrier mechanism for transmitting these streams, RTPtv can also use several unicast streams to service clients that do not have access to a multicast-enabled network. Many different mechanisms and algorithms are used to maintain long-term synchronization between audio and video. In addition, loss concealment and error recovery techniques are used to produce the best possible experience.

This report describes the design and implementation of RTPtv. It is organized as follows. Section 2 describes related work, and section 3 presents an overview of RTPtv. The video components are discussed in detail in section 4, while section 5 is devoted to the audio components. Section 6 explores synchronization issues, and section 7 presents some experiences learned during the development of RTPtv. Section 8 explores various avenues for future work, and section 9 concludes the report.

2 Related Work

RTPtv uses MJPEG for transmitting video. The *JPEG* format [ISO, 1991] [Wallace, 1991] is commonly used for the efficient storage and representation of still images. The term MJPEG indicates that a sequence of JPEG images are used to represent video. *MPEG* [ISO, 1993] [ISO, 1994] is another format that can be used for representing video. Like JPEG, MPEG compresses individual video images. However, MPEG supports more sophisticated compression techniques for intra-frame compression, such as dividing an image into regions and processing each region using a unique set of compression parameters. In addition, MPEG supports inter-frame compression to take advantage of similarities between multiple frames.

Although MPEG is typically more efficient than MJPEG for transmitting video, there are tradeoffs in using MPEG over MJPEG. MPEG usually takes more time to encode frames, resulting in increased end-to-end latency. This increase is due to the fact that MPEG uses inter-frame compression, and thus an MPEG encoder must wait for multiple video frames/fields to be captured before compression for a specific frame can be completed. When decompressing the video stream, certain key-frames must be decoded before other frames can be processed. As a result, MPEG generally results in a higher end-to-end latency in television transmission. In the case of interactive videoconferencing using MPEG, this latency can be too large for interactive two-way communication.

MJPEG is less expensive then MPEG, both in terms of financial and computational costs. Consequently, MPEG offers lower bandwidth for the same quality but higher end-to-end latency and encoding cost. Moreover, MJPEG typically behaves better when data loss occurs since certain types of loss in an MPEG stream can affect multiple video frames. There are several commercial MPEG implementations that perform a function similar to RTPtv, but these systems have a much higher cost (generally $3-5K) and typically support only MPEG-1 (which is limited to a video image that is one-quarter the size of conventional television). Systems based on MPEG-2 which support full-sized video are more expensive ($5-30K). RTPtv supports full-sized video, and costs less than $2K: a Linux-based computer and an inexpensive MJPEG video card ($400) are all that is required. When MPEG encoders become more cost-effective, the contributions of this report will be equally useful for implementations based on MPEG hardware.

Low cost software-only implementations of RTP audio/video applications also exist, but they are limited by the computational performance of today's processors and cannot produce the same quality as RTPtv. Although some research applications, such as *rat* [Hodson and Perkins, 1999], can produce audio that is the same quality as RTPtv, the video quality is quite inferior. The *RealPlayer* [Real, 2001], *Windows Media Player* [Microsoft, 2001], and *QuickTime Player* [Apple, 2001] are three popular commercial software implementations for viewing multimedia streams. Neither the *RealPlayer* nor the *Windows Media Player* are RTP-compliant, and none of these implementations appear to be capable of operating under low-latency conditions. In addition, these programs do not support certain required

features of RTP (such as the RTP extension header), nor do they support certain standard codecs such as the *Redundant Audio* codec [Perkins, et al. 1997].

3 Overview of RTPtv Design and Implementation

This section presents an overview of RTPtv and some of the features offered by the system, such as synchronized playback, the use of RTP, and the ability to use unicast and/or multicast transport. Before describing RTPtv in detail, it is useful to provide an overview of the software and hardware that comprise the system. RTPtv was implemented using C++, and uses Tcl/Tk for the (optional) GUI user interface [TclTk 2001]. Figure 1 shows RTPtv in use. The television monitor on the left is showing a television broadcast that is being received via RTPtv. The RTP GUI can be seen in the computer monitor on the right ride of the figure. The video output can also be displayed in a window on the computer monitor.

Figure 1: Example photo of RTPtv being used to watch a Television Broadcast.

3.1 RTPtv Overview

The overall process of the RTPtv software is to obtain audio and video data from the hardware, fragment the data into RTP packets, transport the packets across a network, receive the packets on a client machine, reassemble the data, and playback the audio/video via hardware. The software implementation is far from trivial, as it uses a variety of algorithms and techniques for maintaining audio/video synchronization, concealing data loss, and handling error recovery.

RTPtv consists of four components: audio server, audio client, video server, and video client. Each of these components exist as a separate command-line program, but they can also be used together within a single-threaded GUI (i.e., several audio and/or video components can run together within the same GUI). The four command-line programs are ideal for situations where a computer has a single, well-defined role as either a server or a client. In situations where a computer is to be used as both a server and a client (such as during a video conference), the GUI interface is a more suitable mechanism for controlling the

system. RTPtv also features a *Remote Procedure Call (*RPC) interface that allows a service to be started, stopped, or adjusted remotely. This interface is described in the appendix.

In terms of implementation, it would be trivial to combine, for example, the video server and video client command-line programs together to form a single video program, but this change would add complexity to the command-line interface—the implementation of a client is not a mirror reflection of a server. They in fact are very dissimilar, so there is also an implementation design argument against combining the client and server video components into a single command-line program. Each program was developed using an event-based programming model, and thus various services can be combined together within the same process (as is the case for the GUI). However, in situations where robust service is desired it can be beneficial to run each component as a separate process so that a transient hardware or software error in one component does not affect another service running within the same process/thread.

The current implementation of RTPtv is intended to operate on Linux-based systems, although the audio components will run on any OS, such as FreeBSD, that supports the OSS API [OSS, 2000]. While RTPtv can use any Linux-supported sound card to capture and play audio, it requires an LML33 card for video [LML33]. The LML33 card is produced and marketed by Linux Media Labs using components and a reference hardware design from Zoran [Zoran, 1997]. The LML33 card is like other video capture cards, but it also contains a processing chip that can be used to encode and decode JPEG in hardware. The peak throughput for the LML33's JPEG codec chip (a Zoran ZR36060) is listed at 29.5Mbtyes/second, so there is a limitation to the data throughput of the board.

When the capture mode is initiated on the LML33, the incoming video signal is encoded as a stream of JPEG images, where each image represents a single video field. The LML33 can also decode a stream of JPEG images, in which case the video can be displayed in a window on a computer monitor and/or output as an NTSC (or PAL) signal that can be displayed on a conventional television. The board has input and output ports for both composite video and s-video.

The source code for the LML33 Linux OS driver is made available via GPL license, but this is both a blessing and a curse. By having the source code available it has been possible to add new features and make desired changes to the driver. However, the driver is only available for Linux, and since the driver is largely unsupported it was necessary to devote a great deal of time to fixing bugs and making the driver more complete. For example, support for `select()` and `poll()` system calls was added, and the driver was enhanced to provide better support for multiple LML33 boards (a system can handle four boards simultaneously). Many changes were also made to the driver to make certain hardware features available to software applications.

Although RTPtv only runs under Linux, other RTP-compliant software, such as UC Berkeley's *OpenMash* [OpenMash, 2001] and Apple's *QuickTime Player* [Apple, 2001], can be used to watch the transmissions at a lower fidelity, due to missing features and the computational limits of CPU processors. For NTSC D1-sized video images at 10Mbps, QuickTime 5.0 is able to display approximately 30 fields/second on a 733Mhz Pentium III dual-processor Windows 2000 machine, while OpenMash 5.1.4 can display up to 30 fields/second on an 600Mhz Pentium III Linux machine (kernel version 2.2.19). OpenMash 5.1.4 only attempts to render the odd-fields of a video stream, and thus it is not possible for OpenMash to exceed 30 fields/second. At 15Mbps, QuickTime can only display about 20 fields/second, and OpenMash can display approximately 26-28 fields/second. When using RTPtv with other RTP applications it is often necessary to disable certain RTPtv features that, although conformant to the RTP standards, are not supported by these applications.

3.2 Synchronization Overview

The topic of synchronization is one that is commonly associated with multimedia research. However, the actual goal of high quality audio/video synchronization is often ignored and rarely achieved. Software tools such as *vic* and *vat* do not make any attempt to synchronize audio and video. These tools run as separate processes, and although this separation may be one explanation for why there is no

synchronization, it can also be argued that synchronization is not provided because these tools use a quality of audio and video that is too poor to make the lack of synchronization apparent.

The University College London (UCL) released a customized version of *vic* that works with a locally produced audio tool, called *rat*, to achieve synchronization. Their approach to synchronization relies on inter-process communication (IPC), and makes certain assumptions about how sound cards operate [Kouvelas, 1998]. These applications were designed to use a multicast session (with a time to live [TTL] of 0) as an IPC mechanism. Although this approach was probably selected because it is largely platform independent, it actually will not work on operating systems that do not provide multicast "loopback" communication (i.e., data transmitted from an application process running on a particular machine will not be received by any other process running on that same machine). The use of IPC can also introduce new coordination problems, since a message that says "start playback of data with this timestamp" might not be immediately received due to delays caused by the OS scheduler (and possible packet loss within the OS).

The UCL developers, like many others, recognized that audio sound cards can vary in terms of the accuracy of their clocks. Although the developers implemented a solution to help address this problem, their solution assumes that a sound card captures audio at exactly the same rate that audio is played. This assumption is actually not correct for a large majority of sound cards today. As a result, the UCL approach to synchronization operates mostly by chance, rather than design.

The general concept for achieving synchronization is to buffer a set of streams, index into those individual streams at points that correspond to each other in time, and then begin playback. The assumption under this model is that synchronization can be maintained with no further effort, as long as the playback process is not disrupted by buffer underflow caused by excessive network data loss. If underflow does occur then the streams can be rebuffered and the synchronization bootstrap process can be executed again.

In reality, this approach to synchronization may work for a short period of time, but the deviation between streams will increase over time and eventually reach a point where the streams appear unsynchronized. The increasing deviation between streams occurs because several clocks are used in the end-to-end process, and each clock varies in speed. For instance, the sound cards on each sender and receiver regulate their sampling rate using internal clocks that operate at slightly different rates. In addition, the system clock on each machine will deviate from other system clocks, as well as the sound card clocks. It is this deviation between clocks that eventually causes problems, both in terms of lost synchronization and buffer underflow/overflow. If a clock on a server is faster than a clock on a client, a gradual buffer increase will occur on the client. Conversely, if the client clock is faster, a buffer underflow will occur since the client is consuming data faster than it is being received. In addition, since video is being processed at a different rate than audio, the result will be a loss of synchronization.

Many software-only audio/video clients attempt to address the effect of clock skew on synchronization by monitoring the amount of delay in the OS audio buffer, and using this information to decide what video frame to display, and when. The net effect is that the video stream is regulated/synchronized based on the sound card clock, rather than the system clock. As a result, the two media streams should remain synchronized because they share a common clock. Although the sending and receiving processes will not share the same clock, the consequences of this are masked by buffering several seconds of data on each client, thus allowing the client to consume data at its own desired rate. Despite using these techniques, many software players still lose synchronization (presumably due to an improper or incomplete implementation). Furthermore, in the case of multicast, or other situations where a client is unable to specify/affect the rate of data delivery, the storage buffers on the client may eventually starve or overflow (depending on how "fast" the client sound card clock is compared to the sender sound card clock).

In the case of RTPtv, the task of achieving media synchronization is even more complex because there are three clocks on each machine (i.e., the LML33 clock, sound card clock, and system clock) rather than two (i.e., the sound card clock and system clock). The three clocks must be unified to a single clock to maintain media synchronization. In addition, the requirement for long-term synchronization, and low

latency for interactive communication, requires that the unified sender and receiver clocks operate at the same rate—if different clocks are used then the client buffers will eventually starve (the occurrence of which will be accelerated by the relatively small buffers) or overflow (if not increase to a size that precludes interactive communication due to delay caused by excessive buffering).

At first the requirement of synchronized clocks on the sender and receiver may seem like a stringent requirement. In actuality, this requirement is easy to fulfill using the *Network Time Protocol* (NTP) [Mills, 1992]. Experimentation has shown that NTP is not required for non-interactive sessions that last for about an hour or less, since the amount of buffering that is used by the client in a non-interactive session can mask any differences between the system clocks. Interactive sessions generally require the use of NTP since the minimal buffers used by the client are prone to frequent occurrences of buffer underflow or overflow due to the tight constraints on delay (and thus buffer size).

Since NTP is the ideal protocol for synchronizing system clocks, it was decided that the LML33 clock and the sound card clock should be unified with the local system clock. As a result, the sending and receiving processes in RTPtv constantly monitor the behavior of the hardware and make occasional adjustments so that the three clocks remain unified. In the case of video, the LML33 board captures at 29.97 frames/second, but since each video frame is a discrete piece of data that is timestamped by the system clock (rather than the LML clock) there is no additional effort needed to unify these two clocks.

Audio, on the other hand, is much more difficult because the audio data is presented as a continuous stream of data without any sort of time information. As a result, the clocking rate of the card must be inferred in software (using a technique described later in this report). To correct for clock inaccuracies, the sender or receiver process may occasionally add or drop audio samples (depending on the card's behavior).

3.3 Extracting useful Synchronization Information from RTP

The RTP protocol is frequently cited as a protocol that is well suited for synchronization. However, few people are aware of the actual effort that it takes to receive the requisite timestamp information from an RTP data stream. Each network packet in the data stream is prefixed with an RTP header. Located within the RTP header is a *media timestamp*. This media timestamp is a 32-bit unsigned integer that advances at a predetermined rate—for video this rate is usually 90,000/second, while for audio the timestamp advances at a rate equal to the sampling rate of the audio capture device.

Taken by itself, this RTP timestamp only provides information on when the data for one packet was captured relative to other packets in the stream. In the case of video, where multiple packets are required to transmit a single video frame, the timestamp can also be helpful for identifying which packets correspond to the same video frame. It is nearly impossible to synchronize multiple media streams without having a reference to a wall-clock for each of those streams (it is also helpful if each of the streams use the same clock, but that is a separate matter discussed later). The RTP stream, by itself, does not provide information to map the timestamp integer to a wall-clock time. To derive a wall-clock time from the RTP media timestamps, information from the companion RTCP stream must be used.

An RTP transmission actually consists of two network streams: an RTP stream and an RTCP stream. Media content is transmitted in the RTP stream, while the RTCP stream is a control stream used by clients and servers to exchange status reports. Each server transmits *sender reports* that provide status information about the RTP stream, while each client transmits *receiver reports* that describe how well (or poorly) the RTP stream is being received.

One important piece of information that is transmitted in each RTCP sender report is an NTP timestamp of when that report was transmitted from the server. In addition, the RTCP sender report includes a second timestamp, in the form of an RTP media timestamp, that is equivalent to the NTP timestamp. It is these two timestamps that allow a client to derive the wall-clock time for when a piece of data was captured. For example, suppose that a video client receives an RTP packet with a media timestamp of z, and that the client also receives an RTCP sender report with NTP timestamp x and RTP

media timestamp *y*. The media timestamp in the RTP packet can then be converted to a wall-clock time by calculating *(z-y)/90000* and adding that result to the NTP timestamp *x*.

This explanation ignores certain issues, such as conversions to/from NTP format and situations where the RTP media timestamp has returned back to zero due to overflow, but it provides a basic example of how each RTP packet can be calculated as an offset to a wall-clock time specified in an RTCP sender report. The actual conversion of NTP to an RTP media timestamp is relatively straightforward, but the issue of overflow and underflow does complicate the conversion of an RTP media timestamp to NTP. The conversion process involves some floating point arithmetic, but experimentation has shown that it would take several months or years before arithmetic error would compound to a level that might begin to complicate the situation. Regardless, RTPtv attempts to use integer calculations whenever possible to minimize the potential for error introduced by floating point arithmetic.

Due to the design of RTP, a client must monitor four different network streams to provide synchronized audio and video: a video RTP stream, an audio RTP stream, and two RTCP streams. The management of these four streams can be quite complicated, especially for interactive environments with low-latency, since it is necessary to examine the contents of all four streams. In addition, it is also necessary to examine and track the arrival times of packets in the RTCP streams. The arrival time of RTCP packets is important for calculating accurate clock skews—the monitoring of clock skew is not as vital when NTP is used on all machines, but the technique is still important for estimating the packet propagation time within the network.

The RTCP stream is designed to be a low bandwidth connection. Although the RTP specification describes algorithms for transmitting RTCP reports in a scalable fashion over multicast, many RTP implementations transmit a report at a fixed interval of five seconds. Since a sender report may only get sent every five seconds, this means that a client may need to wait a few seconds for a sender report to be received before it can start a synchronized playback of the media.

RTPtv attempts to address this problem by transmitting extra sender reports whenever a new client is detected. Three copies of an RTCP sender report are sent when a new unicast client is detected, while only a single report is sent for multicast. This discrepancy between the two media types exists out of a concern for the scalability of multicast—it is unlikely that scalability of this technique will ever be a problem unless several dozen clients all joined the session simultaneously. The RTP specification describes an algorithm for determining an appropriate report transmission rate. The algorithm is based on the bitrate of the RTP stream, and in the case of RTPtv the algorithm specifies that a report can be sent multiple times per second. This algorithm appears to have been designed with lower bitrates in mind, which is one reason why RTPtv has been configured to transmit reports at five second intervals.

The extra RTCP reports sent by the servers usually eliminates the need for a client to wait for a sender report to be received, but it does not eliminate the possibility that the extra reports may get dropped within the network. Our experience with RTPtv has shown that this type of packet loss is a rare occurrence, and that there is only a short delay (at most a few seconds) before the client is able to receive a sender report and playback a synchronized stream.

3.4 Unicast and Multicast Transmission Modes

Most RTP-compliant applications support only one type of network stream at a time: either unicast or multicast. Although some applications have support for both stream types, they can only use one stream type at a time. However, the servers used by RTPtv are capable of using both stream types simultaneously. Each stream type has its own advantages: multicast is an efficient way to transmit to several clients simultaneously, while unicast can be used for private communications or to transmit to machines that can not receive a multicast stream. In addition, unicast can be preferable because not all networks are capable of handling a multi-megabit multicast stream, and some networks broadcast multicast packets to all hosts within a switch fabric, which can result in decreased performance for other machines using the switch fabric [Peterson and Davie, 2000].

Because a 10 or 20Mbps stream can overwhelm some networks, care was taken to regulate when servers transmit packets. After a unicast stream has been initiated, the server will continue to transmit that stream to the client until one of the following occurs: 1) an RTCP "bye" packet is received, 2) the client OS uses ICMP [Fenner, 1997] to refuse a packet, or 3) it has been 60 or more seconds since an RTCP receiver report was received from the client. The 60 second timeout was originally 25 seconds in duration; 25 seconds was selected because the RTP specification suggests that a client can/should be expired after 5 RTCP report intervals (and 5 reports intervals is approximately 25 seconds). When using RTPtv we found that a 25 second duration can sometimes cause an active client to timeout because the network dropped the most recent RTCP receiver reports from the client. These unintended timeouts are much less frequent when an interval of 60 seconds is used. The timeout mechanism in unicast streams is only useful when a client is using an OS that does not send ICMP "refused" messages for unwanted UDP packets.

In keeping with the "architectural principles of the Internet" on the parsimonious use of unsolicited packets [Carpenter, 1996], RTPtv servers only transmit multicast data when one or more clients are detected in the multicast session. In contrast, most multicast servers transmit continuously, regardless of whether there are any clients to the stream. The conventional belief is that it is not harmful to transmit data when no clients exist since the data stream will only be delivered to network paths that lead to a client and, since no clients exist, the stream will terminate in the network at a location that is close in proximity to the server. Under ideal circumstances, this belief is valid. However, if a server could be used for other purposes, or if the server is running on a personal machine, the effort to transmit the stream is a waste of resources. In addition, the "unused" multicast stream may still get propagated down a few links (e.g., to the *Rendezvous Point* of a PIM-SM-based [Estrin, et al., 1998] network), thereby wasting bandwidth and potentially affecting other users if the local switch fabric has been configured to broadcast multicast packets.

To make clients aware of the servers that exist in a multicast session, sender reports are always transmitted at a regular interval even if no clients exist. For a quiescent stream, most of the data in each report will be static but it will still allow a client to determine which (if any) servers exist on a multicast session, as well as to obtain the necessary timestamp information for synchronization. RTPtv can also be configured to multicast a stream continuously, rather than on demand. One argument for streaming continuously is that client detection is based on the receipt of RTCP receiver reports at the server, and if these packet are dropped (or fail to be received by the server due to multicast problems) then a client will be unable to initiate the stream. A possible option for future work in multicast is to add support for muting multicast streams within the operating system of a server (or even to propagate this information to the application level, perhaps by using the `select()` or `poll()` system calls to indicate that a network socket is "write-able").

Another feature of RTPtv, albeit small, is that there exists a mode where RTCP reports for unicast sessions are transmitted on a multicast channel. The intent behind this feature is to allow existing RTP utilities, such as *rtpmon*, to monitor unicast sessions by listening to the "multicast version" of these reports [Bacher, et. al., 1996]. This feature is rarely used, but can be helpful for tracking the performance of a unicast stream over time.

3.5 RTSP Server Support

The client-detection mechanism in RTPtv relies on the receipt of RTCP receiver reports at the server. *RTSP* is an IETF protocol that has been developed for creating, starting, pausing, stopping, and terminating media streams [Schulzrinne et al., 1998]. To better operate with clients that are RTSP-compliant, RTPtv servers support a baseline implementation of RTSP. By using RTSP, a client can obtain information about a stream, initiate a stream, and stop a stream. RealPlayer support for RTP is incomplete, but the QuickTime Player can use RTSP to initiate a unicast stream from RTPtv. When multicast is used there is no need to use RTSP—the stream is already being transmitted and the session information is obtained via *SAP* [Handley et al., 2000] and *SDP* [Handley and Jacobson, 1998].

To initiate an RTPtv stream using an RTSP-compliant client a user specifies a URL such as rtsp://audioserver:7700/audio or rtsp://videoserver:7800/video. The specifications for SAP, SDP, and RTSP provide different mechanisms for describing and initiating a media stream. Experience with RTPtv has shown that there is an increased chance of successfully inter-operating with other implementations when the audio and video streams are specified (and made available) individually, rather than combined. In addition, the SDP description for the media stream should be available via both SAP and RTSP, even if the steps have been taken in one protocol to eliminate the need for using the other.

The following sections will describe, in detail, certain aspects of the sending and receiving processes. After this review the topic of synchronization will be re-visited to provide an end-to-end perspective on how these aspects work together to maintain synchronization.

4 Video Client/Server

This section describes, in detail, aspects of how RTPtv transmits and receives a video stream. The design issues of the video server will be described first, followed by the video client.

4.1 Video Server

In RTP transmissions, the fundamental role of a server is to execute a relatively straightforward event-based cycle: wait until data has been captured by hardware, process/format the data, transmit the data, and repeat. The video server supports transmission of interlaced D1 images at 60 fields per second, or progressive CIF images at 30 frames per second. The server allows an administrator to specify a particular image quality level or bitrate. In addition, a custom frame-rate can be specified. The server can also be commanded to switch between live video and a static, user-provided JPEG image.

The creation of the video server involved some interesting design issues, both in terms of the interaction with the video capture device and compatibility with the RTP JPEG specification. The following sections discuss several design issues, such as marking video frames with timestamps, processing the JPEG quantization tables, handling the limitations of the RTP JPEG specification, and determining the quality scaling factor used by the LML33 board.

4.1.1 Timestamps of Video Frames

One common problem mentioned by previous researchers is that it can be difficult to obtain an accurate timestamp for when a video frame is captured—accurate timestamps are important for clients, since these timestamps are used to schedule playback of frames and/or perform synchronization. Video capture drivers do not typically provide timestamp information, so the timestamp is generally obtained by requesting the current time immediately before or after the system call that reads the video frame. The result of the system time request is then assumed to be an accurate reflection of when the video frame was actually captured. In essence, the software assumes that the execution time of the system call is either constant or bounded by an acceptable synchronization tolerance.

For low-quality applications, the accuracy of timestamps is not a huge concern, but it can be a significant problem if all the storage buffers in the OS driver are full and a decision must be made between overwriting the oldest occupied buffer, or halting the video capture process. In situations where an old buffer is reused to store the most recently captured frame, it can be difficult or impossible for a video application to know if a particular frame is missing (because it has been overwritten). However, even in a normal situation it can be impossible to know exactly when a frame was captured if the driver does not provide a means for reporting how many frames are buffered in the driver. If several frames are buffered in the OS, the timestamp procedure described above will provide the wrong timestamp since the solution assumes that each frame is the most recently captured image (when it was actually captured one or more frame-times in the past).

In the early development stages of RTPtv we encountered a similar problem in obtaining accurate timestamps. Although the LML driver provides a timestamp in the application header of the JPEG frame, each frame is marked with a timestamp as part of the call to `read()`. This technique is not significantly different from the timestamp procedure that is performed by software at the application level. When using this technique, the timestamps provided by the driver were noticeably skewed from the actual values, especially when buffer reuse occurred due to overflow. Even when a system was lightly loaded, the OS scheduler still caused a significant amount of skew to be observed, so much in fact that a non-trivial corrective algorithm would be required to address the problem. Rather than implement a complex algorithm to handle and interpret these (potentially) inaccurate timestamps, the OS driver was modified so that each video frame is marked with a timestamp by the driver's interrupt handler. As a result of this change there has been a significant increase in the accuracy of the timestamps, and the issue of frame timestamps is no longer a problem because each video frame is marked with the exact time that a frame was captured.

4.1.2 JPEG Quantization Tables

Another important issue deals with the JPEG quantization tables. The LML33 board supports two bitrate modes: dynamic bitrate and fixed bitrate. The dynamic bitrate mode uses a single fixed JPEG quantization table (appendix K.2 from the JPEG specification); which results in a bitrate that varies according to the content of the video images being captured. Experimentation showed that this bitrate can vary anywhere from 3 to 13Mbps—6 to 10Mbps is typical, but commercials almost always take more bandwidth than the actual television show.

In contrast, the fixed bitrate mode bounds the bitrate by selecting a different quantization table for each video field. The LML33 board selects an appropriate quantization table by examining the amount of data that was produced for the previous field and uses that information to adjust the quantization table scaling factor that is used for the next frame. An unfortunate consequence of this approach is that the LML33 board uses the same data bitrate regardless of whether a particular video image could be encoded at a lower bitrate. A more useful hardware feature for RTPtv would have been to permit a maximum bitrate to be specified. Under this model, a dynamic bitrate would be used unless that bitrate exceeded the specified threshold—only then would the quality be adjusted to prevent an excessive amount of bandwidth from being used. The following subsections discuss some of the issues that were dealt with in regards to how the LML33 board uses quantization tables.

4.1.3 RTP JPEG Header

Each JPEG image consists of image data that is preceded by one or more (typically several) JPEG headers (see Figure 2). These headers provide information about the image data, such as the size of the image and the methods used to encode the image. The RTP JPEG specification was designed so that in most cases it is unnecessary to transmit JPEG headers, including quantization tables, in the RTP video stream. To achieve this goal the specification was written so that only particular types of JPEG images can be used, thus making it easier to concisely report the information that would otherwise have been included in the JPEG headers.

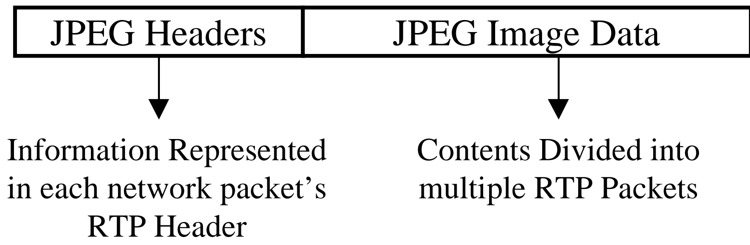


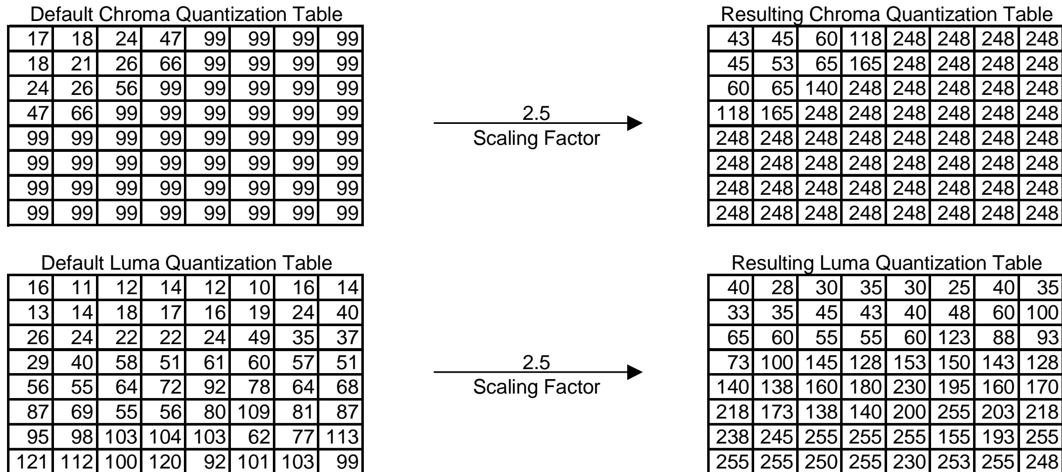
Figure 2: High-level depiction of a JPEG Image.

Header	Fields (bits 0–31)					
IP	Version	Hlen	TOS	Length		
IP	Ident	Flags	Offset			
IP	TTL	Protocol	Checksum			
IP	SourceAddr					
IP	DestinationAddr					
UDP	SrcPort	DstPort				
UDP	Checksum	Length				
RTP	Ver	P	X	CC	M	PT, Seq
RTP	Ts					
RTP	ssrc					
RTP-EXT	ext. header identifier	ext header length				
RTP-EXT	Q table scaling factor					
RTPJPEG	Type-specific	Fragment Offset				
RTPJPEG	Type	Q	Width	Height		
JPGRSTRT	restart interval	F	L	restart count		

Figure 3: Network Packet Headers for RTP JPEG Video (as used by RTPtv).

As part of this conservation effort, the RTP JPEG specification provisioned a single 8-bit value, called "Q", to report which quantization table corresponds with a JPEG image (Figure 3 shows all of the RTP headers that are present in each packet transmitted by an RTPtv video server). The single 8-bit value is a concise substitute for what would otherwise be two 64-byte quantization tables (for a total of 128-bytes). When the dynamic bitrate mode is used, this 8-bit Q value is not an issue since one of its defined values, 50, corresponds exactly with the quantization table used in the dynamic bitrate mode.

However, when the fixed bitrate mode is used, the definition of the 8-bit Q field is too constraining, and visual artifacts are apparent. If we assume that a quantization table must be a linear scaling of the K.2 table (i.e., each value in the table is multiplied by the same real number—see Figure 4 for an example), there are approximately 11,130 different tables that can be used. Since only 99 values have been defined for the Q value in the RTP JPEG payload format, this means that less than 0.9% of the possible quantization tables can be exactly represented using conventional RTP mechanisms.



Default Chroma Quantization Table

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

Resulting Chroma Quantization Table

43	45	60	118	248	248	248	248
45	53	65	165	248	248	248	248
60	65	140	248	248	248	248	248
118	165	248	248	248	248	248	248
248	248	248	248	248	248	248	248
248	248	248	248	248	248	248	248
248	248	248	248	248	248	248	248
248	248	248	248	248	248	248	248

Default Luma Quantization Table

16	11	12	14	12	10	16	14
13	14	18	17	16	19	24	40
26	24	22	22	24	49	35	37
29	40	58	51	61	60	57	51
56	55	64	72	92	78	64	68
87	69	55	56	80	109	81	87
95	98	103	104	103	62	77	113
121	112	100	120	92	101	103	99

Resulting Luma Quantization Table

40	28	30	35	30	25	40	35
33	35	45	43	40	48	60	100
65	60	55	55	60	123	88	93
73	100	145	128	153	150	143	128
140	138	160	180	230	195	160	170
218	173	138	140	200	255	203	218
238	245	255	255	255	155	193	255
255	255	250	255	230	253	255	248

Figure 4: Example Derivation of Quantization Tables using a Scaling Factor of 2.5.

Consequently, the video server is forced to pick the Q value that most closely corresponds to the quantization tables used by the LML33 board to encode the JPEG image. The board does not indicate the scale factor used to encode a particular video field, rather it just gives the quantization table. Consequently, the server must infer the appropriate scale factor by analyzing the quantization tables, and then pick an appropriate RTP Q value. In order for a receiver to correctly render a JPEG frame, accurate quantization tables must be used. Since the receiver uses the Q value to recreate the quantization tables, the resulting tables only roughly approximate the tables used to encode the image. The following chart (Figure 5) shows the sum-of-squares error for the various scaling factors that can be used by the LML33 board, where each line represents a different scaling factor range starting at 0.0 and ending at a particular integer scaling factor. For example, the lowest line ranges from 0.0 to 1.0, while the highest line is 0.0 to 10.0. Because of these scaling errors, visual artifacts are noticeable on playback.

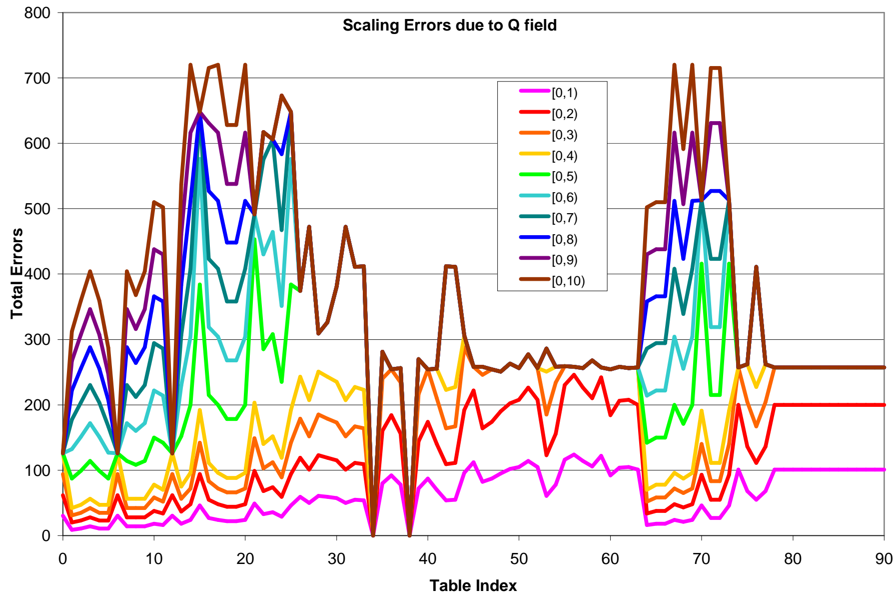


Figure 5: Sum-of-Squares Error caused by the RTP Q field for various Scaling Factor Ranges.

These visual artifacts appear in two different ways. The first, and most noticeable, artifact is the appearance of flashes. These flashes occur because the LML board can use a different quantization table for each video field. The quantization table error for each field will vary based on the magnitude of the error in the Q value, and when the error increases or decreases by a significant amount then a quick transition from a slightly dim image to a slightly bright image (or vice versa) causes the appearance of a flash. These flashes are not as apparent at certain bitrates because the LML board is less constrained by bandwidth and less likely to change to a different quantization table during the same "scene" in a video.

Figure 6 shows the scaling factors for all the Q values (1-99) that are statically defined by the RTP JPEG specification (the distribution for these values is shown in Figure 7). The scaling factors in Figure 6 are shown on a logarithmic scale to make each value easier to discern. The figure shows that a majority of Q values result in a scaling factor that is very close to the value of 1.0; there are relatively few choices for Q values when using scaling factors that are significantly larger or smaller than 1.0. A small Q represents a low quality image, and the small selection of scaling factors is typically not an issue because the poor quality serves to mask the errors. However, the limitations of the Q field are a bigger problem when the LML33 board is used to encode high quality images. For bitrates of 24Mbs or higher, the resulting images usually require a Q value of at least 90. Because of the way in which the Q values are scaled, there are very few Q values to choose from when encoding at the higher-quality. Consequently, in some situations high bitrate streams (>24Mbit/sec) can look inferior to a lower bitrate stream (e.g., 15Mbit)—this situation is entirely due to the design of the Q field in the RTP JPEG header.

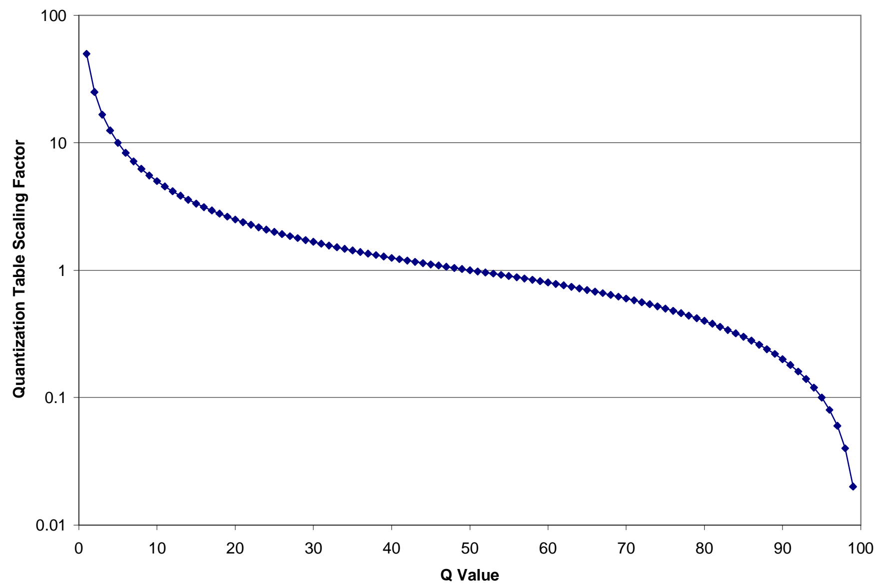


Figure 6: Quantization Table Scaling Factors for the Statically-Defined Range of the RTP JPEG Q Field.

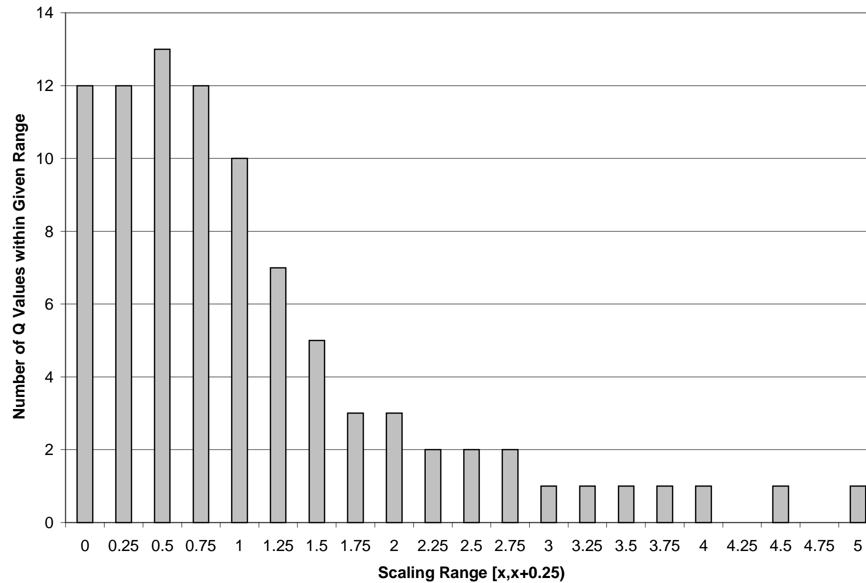


Figure 7: Distribution of the Quantization Table Scaling Factors for the Statically-Defined Range of the RTP JPEG Q Field RTP JPEG Q Field.

The second type of visual artifact is the appearance of blocks in the video image due to the fixed size block transform used in JPEG. These blocks mostly occur in cartoons or commercials that display a company logo on a plain white background. For example, if the Japanese national flag is displayed on the screen, the large red dot will transition between having smooth edges and blocky edges depending on what other objects, if any, are also displayed on the screen.

The problem of representing a quantization table in an RTP packet is an important issue to address. Although the RTP specification does provide two different methods for using a custom quantization table, neither seems appropriate for high-quality MJPEG streams. The first method uses the Q range 127-254, in conjunction with a special *quantization table header* for defining new JPEG quantization tables. The RTP specification requires that once a table is assigned to a particular Q value, no other quantization table can be used for that Q value. Even if every Q value (0-255) were used to represent a custom quantization table, only 2.3% of the possible 11,130 tables can be used. Consequently, an 8-bit value is insufficient for fixed bitrate transmissions. As an experiment, the 127-254 range was assigned to tables that fall between the defined 1-99 range such that 126.5 would be subtracted from the value and the result evaluated using normal mechanisms (e.g., 167 would be treated as if it were 40.5, and 168 would correspond to 41.5). Unfortunately, this technique offered little (if any) improvement.

The second mechanism that RTP provides for specifying a custom quantization table is similar to the first, except that the value of 255 is not assigned to any particular table, and basically serves as a way to include a unique quantization table with every jpeg frame. Unfortunately, this mechanism has some undesired tradeoffs. For example, two 64-byte quantization tables must be included with every video frame, and since these tables are only included in the first packet of a video frame then a client will have no idea as to which quantization table to use if that first packet is not received (since Q is set to 255). The client could just re-use the table from the previous frame, but this will in all likelihood result in visual artifacts.

Since neither method is acceptable, a third alternative was selected. This method involves using the RTP *application extension* header. This header was designed to allow custom extensions to be used without being formally incorporated into the RTP specification. This approach has proven to be a viable solution, but it does have some tradeoffs. The first tradeoff is that 8 bytes of packet header overhead are

now introduced into the stream. Although 8 bytes can be considered substantial, it is minor when compared to the 128-byte size of the quantization tables.

To assist in compatibility with other RTP applications, the use of the application extension header is an RTPtv feature that can be disabled. Another tradeoff for using the application extension header is primarily philosophical, in that storing JPEG-specific information outside of the RTP JPEG header can be considered a violation of the encapsulation principle by which most Internet protocols are designed (however, this is not the only exception—the CRC field of a TCP header includes portions of the IP header).

An RTP application extension header has a defined structure for specifying a profile type of the extension header and the amount of the data contained within the extension header. A profile type of "77" was arbitrarily selected, and the minimum data length of one word is used. In keeping with other Internet specifications, it was decided that this word should represent the scaling factor for the quantization table, whereby the structure of that word is defined to be a 32-bit fixed-point number with 16 bits on either side of the decimal point. The precision of this scaling factor is excessive for 8-bit quantization tables, but no other useful purpose could be determined at this time for the remaining bits. Also, this level of precision may be useful for implementations that utilize quantization tables with 16-bit values. It should be noted that the Q value of the JPEG header is still set to the appropriate value even when the application extension header is used, so that other RTP implementations can receive the JPEG stream.

4.1.4 Determining Quantization Table Scaling Factor

An interesting implementation problem that needed to be solved was to determine, given a pair of quantization tables, the exact (or nearly exact) scaling factor that was used by the LML33 to compute these tables. This scaling factor is needed so that it can be included in the application extension header, but also so that the appropriate Q value can be derived from the scaling factor.

The hardware specification for the LML33's JPEG chip mentions a 16-bit fixed-point scaling factor register (with 8-bits on either side of the decimal point). At first the solution seemed to lie in reading the contents of the register just after each JPEG frame had been captured. Unfortunately, this solution adds additional system call overhead, and it adds new deadline constraints for reading the register at the appropriate time (which was defined by the chip specification to be just after a capture process).

From experimentation it was determined that the scaling factor register is not always an exact reflection of the most recently captured frame. The LML33 board returns JPEG images in pairs (each pair consists of an odd field and an even field); since each field can have different quantization tables it would be impossible for a single register to reflect the scaling factors used for both frames. As a result, the most accurate solution for determining a scaling factor involves calculating, for each unique value in the quantization table, the minimum and maximum scaling factor that could have been used to obtain that value in the table. The final value is then a numeric range that is bounded by the largest minimum and the smallest maximum scaling factor for all the values in the table.

Although this method is accurate, it is excessively expensive in terms of the number of calculations that are performed. Since this approach would involve, at a minimum, 50 floating-point divisions (a computationally expensive operation), a more efficient method was developed. This solution takes advantage of the fact that the JPEG chip on the LML33 board uses a 16-bit fixed-point number.

For review, a fixed-point number is interpreted (i.e., converted to conventional floating point) by reading the entire number as an integer and dividing that number by the result of 2 raised to the number of bits to the right of the decimal point. So, in the case of the LML33 board with 8-bits on either size of the decimal point, the scaling factor is obtained by treating the contents of the register as a 16-bit unsigned integer and dividing that value by 256.0.

By fortunate coincidence, the "standard" JPEG quantization table contains a number with the value of 64. Since a fixed-point number is used, this means that the corresponding value in the resulting table was obtained by calculating $64*(x/256)$, where x represents the scaling factor as an unsigned integer. Since

computers use binary to represent numbers, this calculation is equivalent to shifting x left by 6 digits, and then shifting that result right by 8 digits (see Figure 8 for a more accurate depiction). Consequently, if you look at a quantization table and find the value that corresponds to the value of 64, and divide that value by 64, then the result is almost identical to x. In fact, the only difference between this result and x is that the resulting value will have the highest 6 bits and the lowest two bits set to zero. Assuming that the value of x is less than 5, the estimated scaling factor will differ from x by at most 3/256.

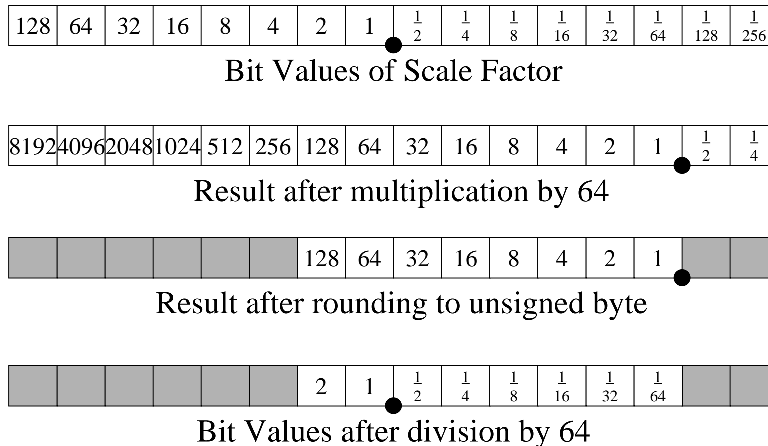


Figure 8: Depicts how the Values in the Scaling Factor change when the value of 64 is scaled, and then that result is used to derived the value of the Scaling Factor.

Although 3/256 is a fairly small value, in experimentation this small degree of error still caused visual artifacts to appear on occasion. To address this problem, the estimated scaling factor is used to calculate a new set of tables. This new set is compared with the original set, and the estimated scaling factor is increased or decreased by 1/256, if necessary. This verification/validation step needs to be performed at most 2 times before the exact scaling factor is obtained. The technique of using 64 only works if the scaling factor is less then 5. Otherwise, the value of 16 is used, instead of 64, and the comparison step may need to be performed up to 8 times since the lower 4 bits, rather than just 2 bits, are masked off when 16 is used instead of 64. The value of 16 is only needed when an extremely low bitrate is specified, and if the scaling factor is greater than 16 then the value of 10 (the smallest value in the table) is used. Only a limited number of correction attempts are made when 10 is used, due to the fact that the video quality will be extremely poor regardless of the accuracy of the scaling factor.

The overall algorithm is not computationally expensive. The tables are created using integer multiplication, and no division operations are used. Although there are a total of 128 values in the two quantization tables that need to be compared, if the tables are represented as a linear array in memory then approximately 50 of the values at the tail of the array can be ignored because they are identical to each other. Each of the values is an 8-bit integer, but it is much more efficient to compare the arrays by treating them as an array of 32-bit integers (in fact, this method is almost twice as fast as using `memcmp()`).

Experimentation has shown that of the four possible situations for correcting a scaling factor, each occurs with almost equal probability: an exact match is found 26.48% of the time, while 24.11% of the scaling factors are increased by 1/256, 24.51% are decreased by 1/256, and 24.90% are decreased by 2/256. Although the probability of an exact match may dominate the other three possibilities, it is hard to ignore the fact that almost three-quarters of the scaling factors will be slightly incorrect if the corrective measures are not taken.

4.2 Video Client

One of the goals for RTPtv was to build a system architecture that could gracefully handle packet loss in an efficient manner. To meet this desire, the system was designed so that 1) each network packet can provide a useful quantity of data without regard to whether a neighboring packet was received, and 2) the loss of a network packet does not render other packets useless. In contrast, when the *vic* video application fails to receive a single network packet for a JPEG frame, all data corresponding to that frame is discarded. A 10Mbps D1 video stream generally requires about 25 Ethernet packets per JPEG image, and thus if a single packet is lost then the other 24 packets for that video field are useless. This behavior exists primarily because it is not possible to process a particular piece of JPEG data without first decoding the preceding portion of the image. If image data is missing, it is not feasible to decode the image region that lies within, or after, the missing data. In fact, *vic* drops the entire video field.

The following subsections discuss two particular aspects of the video client: the use of restart intervals to assist in loss recovery, and scheduling playback of video frames.

4.2.1 Loss Recovery using JPEG Restart Intervals

Since RTPtv uses higher quality JPEG images, each video frame is, by necessity, transmitted using a larger quantity of network packets, and thus each video frame is more susceptible to being discarded in the event that a single packet is lost. To address this problem, RTPtv makes use of a feature in the JPEG specification called *restart markers* (see Figure 9 below). These markers allow an image to be decomposed into several *restart intervals*, where each restart interval represents an equal-sized region of the image. The size of a restart interval is defined in terms of Minimal Coding Units (MCU), where an MCU is defined, for DCT-encoded JPEG images, as an 8-by-8 grid of pixels. Each restart marker signals a point where the computational state for the JPEG encoding/decoding process is reset to a known value, and thus each restart interval can be decoded independently of any other restart interval (i.e., each restart interval can be decoded without knowing the contents of any other restart interval).

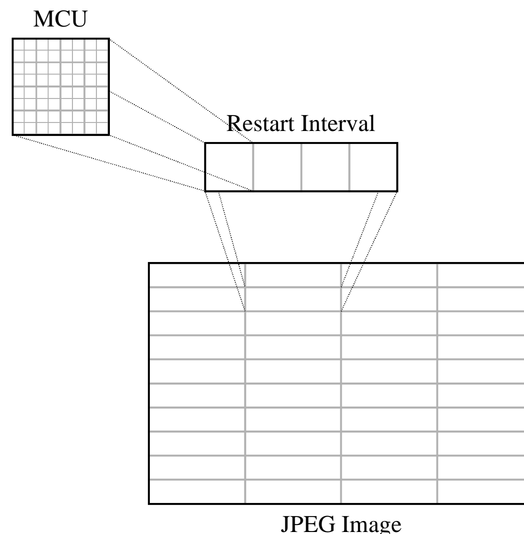


Figure 9: Depicts how a JPEG image can be divided into a grid of Restart Intervals. In the above example, the JPEG image is divided into 40 Restart Intervals, each of which contains four MCUs.

The use of restart markers makes it possible for RTPtv to gracefully cope with network packet loss. If a network packet is lost, then it is still possible for RTPtv to decode the data for any restart interval that is received in its entirety. Because a network packet MTU is relatively small (typically 1500 bytes for most networks), a decision must be made on how to frame the JPEG image data with regard to the restart intervals. This task of framing the JPEG image data is more generally known as *application level framing* [Clark and Tennenhouse, 1990]. The size of a restart interval is defined in terms of pixels, and not bytes, and thus the data size of each restart interval can vary (e.g., experimentation has resulted in intervals that range anywhere from a few bytes to thousands of bytes).

To minimize the negative impact caused by a lost network packet, RTPtv never packs a single restart interval into multiple packets. In other words, each network packet holds one or more complete restart intervals—the quantity of restart intervals in a network packet depends on how many restart intervals can be packed into a network packet without exceeding the MTU of the network (see Figure 10 for an example of how RTPtv performs application level framing).

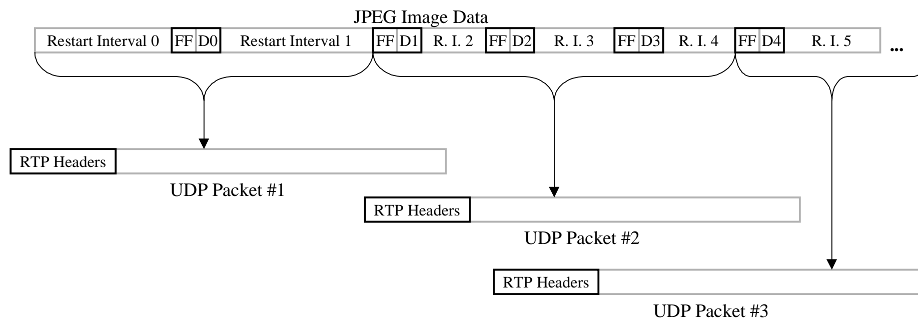


Figure 10: Depiction of how the Restarts Intervals (and the Restart Markers that separate these Intervals) are packed into RTP packets.

The issue of packing restart intervals into network packets presents an interesting question: just how large should the restart intervals be to promote an efficient use of the network? Using large restart intervals will reduce the data overhead caused by the presence of frequent restart markers (and the additional padding of bits to make these restart markers byte-aligned), but large restart intervals also hinder the ability to utilize all of the available space in each network packet.

Generally speaking, the amount of data that can fit in a network packet is reduced by one-half the average size of a restart interval—this is because, when packing restart intervals into a network packet, there is a certain probability that the last restart interval to be placed into a network packet will cause that packet to exceed the network's MTU by one or more bytes (in which case this last restart interval is placed into the next network packet). If we assume that this last restart interval has a 50% chance of fitting within the network packet, then roughly 50% of network packets will be transmitted full, while the other 50% will be sent partially filled.

For example, if 1500 payload bytes can be carried by a network packet, and each restart interval is roughly 98 to 102 bytes in size, then each network packet will carry an average 1450 bytes because half the network packets will be able to hold 15 restart intervals (15*~100 bytes=~1500 bytes), while the remaining half of the network packets can only hold 14 restart intervals because there was not sufficient room for a 15th restart interval (14*~100 bytes=~1400 bytes).

The second problem with large restart intervals is that it increases the possibility that a single restart interval will be too large to fit within a MTU-sized network packet. There are two solutions to this problem: do not transmit the packet, or transmit the packet and hope that the IP layer can reconstitute the oversized packet on the receiver, which will require that the client receive all of the MTU-sized IP frames that comprise that IP packet.

At first this problem may not seem like a significant tradeoff, but if the video image remains relatively static then it is likely that the exact same region of the video image will produce oversized restart intervals on future video frames as well. Consequently, the same region of the video screen will always be at a consistently higher risk for being affected by network loss. Another option to is pack the oversized restart interval so that it overlaps two IP packets at the RTP layer, instead of the IP layer, but this moves an added amount of complexity into the client and does not reduce the severity of the problem. However, as will be described in the following paragraphs, the most effective use of network bandwidth involves using small restart intervals, and thus the problem of oversized restarted intervals is rarely, if ever, encountered.

An experiment was performed to see how the size of restart intervals affects network utilization, both in terms of quantity of network packets, and size of network packets. The results show that the use of small restart intervals results in fewer, and larger, network packets than when large restart intervals are used. Although small restart intervals do result in a higher amount of overhead, this relatively small overhead still provides dividends in packet utilization (e.g., a few bytes of additional overhead might allow 10's or 100's of additional bytes to be utilized in a network packet). To see whether the added overhead of smaller restart intervals was really just making more room available in the packet for the additional overhead, the results were examined in terms of the following parameters: number of packets, size of packets, overall bandwidth consumption, and average number of image pixels stored in a network packet. To clarify, the last examination looked at packet utilization in terms of how much of the image, on average, was stored in each packet—such an examination helps to show the overall effectiveness of each packet (i.e., data throughput minus aggregate data overhead).

One of the first observations from the experiments (shown below in Figure 11) is that the average size of a restart interval is quite well behaved. In the particular experiment used to create this graph, video from CNN (at 15Mbps) was captured using a range of restart interval sizes. The left side of the graph shows the behavior when 1 MCU is used for each restart interval, while the right side of the graph shows the behavior for 100 MCUs per restart interval.

The graph describes three different aspects for each restart interval size: the minimum, maximum, and average number of bytes that were used by the LML33 board to encode the restart intervals. The top line indicates that the maximum size of a restart interval increases quite rapidly as additional MCU are placed into each restart interval, while the lowest line shows that the minimum size of a restart interval is quite small (and remains small even for restart intervals with several MCU). The line for the maximum MCU does not increase monotonically because each trial was run for only a few minutes, and the data obtained from a trial is affected by the characteristics of the video being broadcast while the trial was executing. Regardless, the middle line of the graph shows that the average size of a restart interval increases at a predictable, linear rate as additional MCU are placed into each restart interval. A similar graph is obtained for other bitrates as well, although the large majority of the experiments were used in conjunction with video from CNN.

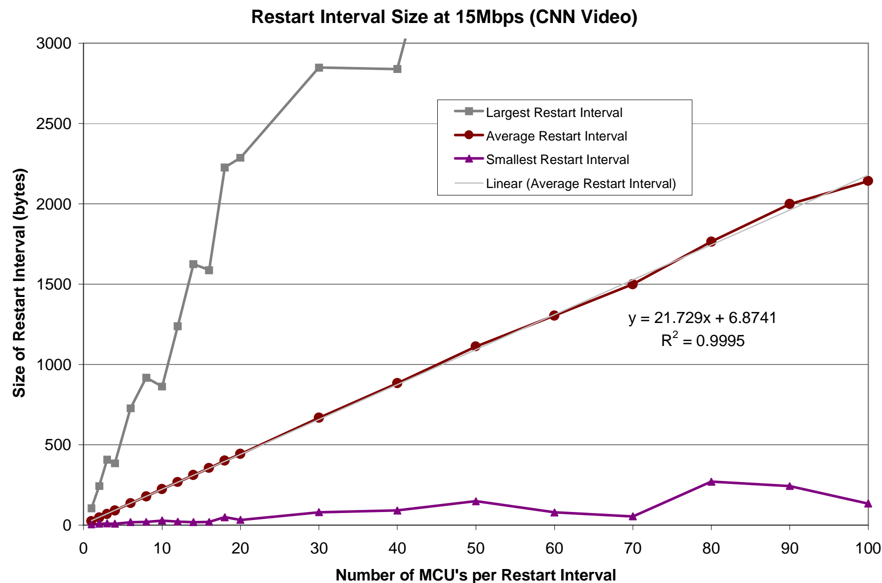


Figure 11: Shows how the Average byte-size of a Restart Interval increases as a factor of the Number of MCUs placed in each Restart Interval.

Table 1 lists “best approximation” equations for the average restart interval sizes at other bitrates, where *x* represents the number of MCUs in a restart interval—the R^2 values for these equations all exceed 0.99. This shows that, for a given bitrate, increasing the image size of each restart interval will increase the size of the restart intervals by a predictable amount. For example, if a stream is running at 4.58Mbps, changing the image size of a restart interval from “g” MCUs to “g+1” MCUs will increase the data size of each restart interval by about 7.2 bytes, while at 9.15Mbps the same change would increase the data size of a restart interval by approximately 14.4 bytes.

Table 1: Estimated number of bytes in a Restart Interval, where “x” represents the number of MCUs in the Restart Interval. Each trial was run for about an hour at various times of the day.

Bitrate (Mbps)	Trial #1	Trial #2	Trial #3	Trial #4	Trial #5
4.58	7.2239x + 4.2334	7.2272x + 4.4074	7.2246x + 4.3598	7.2325x + 4.2543	7.2458x + 3.6393
9.15	14.447x + 6.4818	14.446x + 6.5561	14.455x + 6.3485	14.503x + 6.6218	14.487x + 5.2333
13.73	21.644x + 8.5932	21.711x + 8.4464	21.711x + 8.3409	21.544x + 8.6669	21.729x + 6.8741
18.30	28.913x + 10.84	28.886x + 11.907	28.889x + 11.334	28.319x + 12.201	28.972x + 8.4938

From this table, there is an obvious trend between the slope of these lines and the bitrate of the video stream. The bitrate is specified to the LML33 Board in terms of bits per video field; the rows in Table 1 correspond to 80,000; 160,000; 240,000; and 320,000—multiplying these values by 59.97 (the number of fields per second in NTSC) results in the bitrates listed in the table. For each row in the table, the value of the bitrate divided by the slope of the line, equals approximately 11,078.25 (with a standard deviation of 56.96). This means that, when considering the size of a restart interval, one factor to consider is that the average size of a restart interval will increase by a predictable number of bytes: the number of bits per video field divided by 11,078.25.

Figure 12 shows the average fraction of a JPEG image that is stored in each network packet for various MCU sizes, while Figure 13 shows the average number of network packets needed to transmit a JPEG image. Both graphs share similar properties with experiments conducted at other bitrates, although for lower bitrates the differences made by changing the size of a restart interval are much less significant.

The graphs show that, for the conditions under which the experiments were conducted, a restart interval with 3 MCUs allows for both the highest amount of JPEG data in each packet, as well as the fewest number of network packets. Table 2 lists the optimal number of MCUs for several different bitrates, and shows that the lower bitrates tend to prefer larger restart intervals (i.e., more MCUs in each restart interval) while higher bitrates prefer smaller restart intervals.

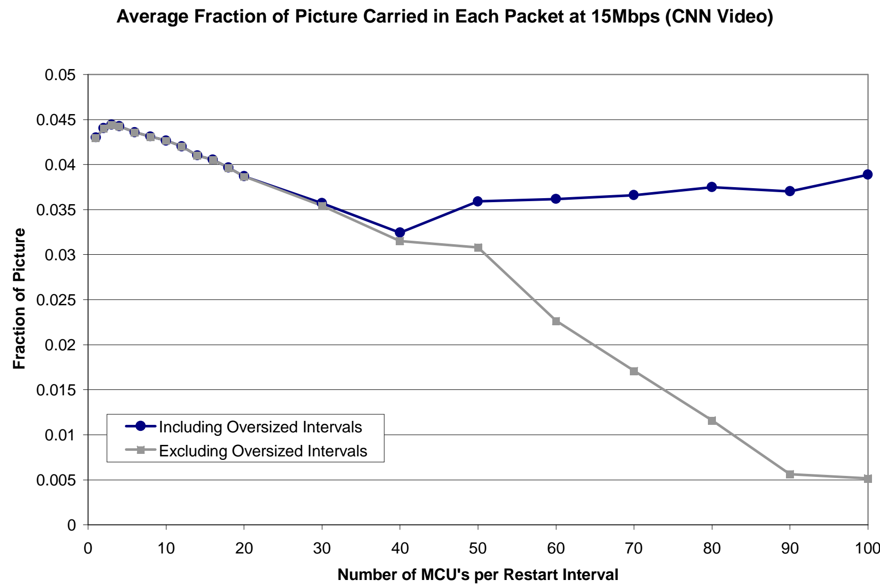


Figure 12: Average Fraction of a JPEG Image that is stored in each Network Packet, as a function of the number of MCUs in each Restart Interval.

Table 2: Optimal number of MCU to use for optimizing Network Packet Utilization at various Bitrates.

Bitrate (Mbps)	Optimal MCU
2.29	6
4.58	3
9.15	3
13.73	3
18.30	2

Using three MCUs per restart interval seems quite small, at least when compared to software programs that typically use tens or hundreds of MCUs per restart interval. However, Figure 14 clearly demonstrates that increasing the size of a restart interval has a negative impact on how many bytes can be utilized in each network packet.

One of the benefits of a small MCU is that there is a much smaller chance of encountering a restart interval whose size in bytes exceeds the MTU of the network. Figure 12 has two lines: the line with the clear downward trend is an attempt to show the impact of restarts intervals that exceed the MTU of the network—for this line each restart interval exceeding the MTU is treated as an empty packet. At 15 Mbps the two lines begin to separate at 30 MCUs, while at 20 Mbps the separation begins at 20 MCUs.

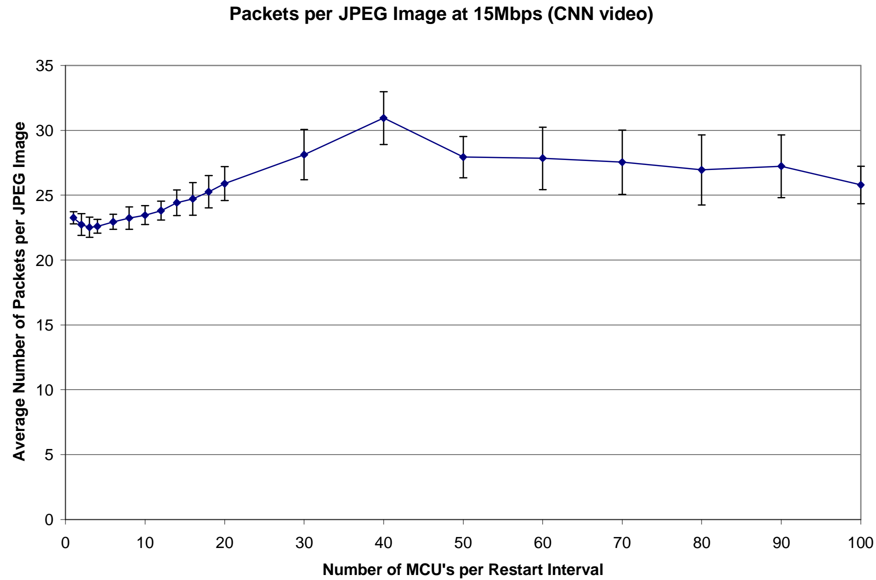


Figure 13: Average Number of Network Packets transmitted for each JPEG Image, as a function of the number of MCUs in each Restart Interval.

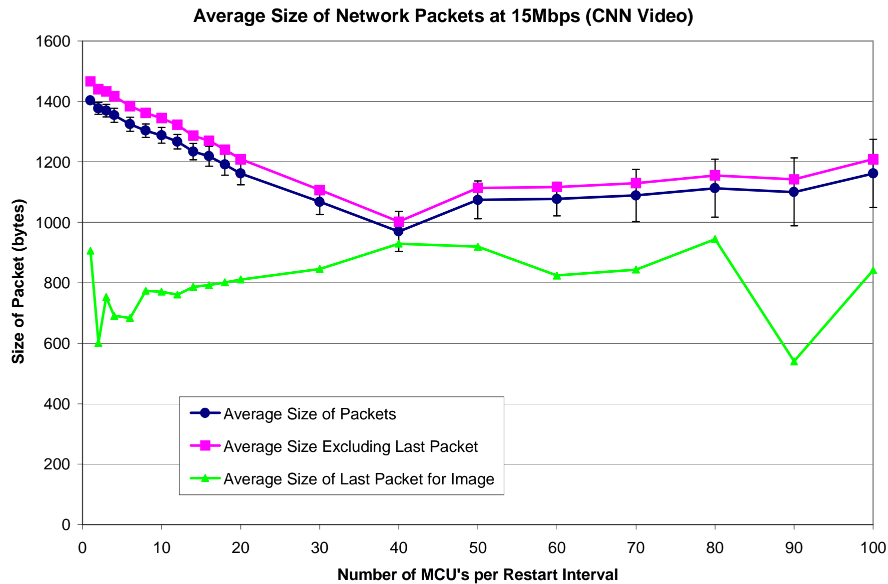


Figure 14: Average Size of each Network Packet, as a function of the number of MCUs in each Restart Interval.

According to these results, the restart marker overhead is almost always offset by the increase in packet utilization, and the decrease in the number of packets (which helps to save on the cost of using 60 byte headers at the start of each network packet). When using fewer than 3 MCUs, the overhead of the restart markers dominates the savings, especially at the lower bitrates. However, one factor that has yet to be considered is the overall bitrate of the stream. Figure 15 shows that when using small restart intervals

(such as 3 MCUs), there can be a slight increase (generally 1 to 2%) in the overall bitrate of the stream, although a 10% increase can result if only a single MCU is used in each restart interval.

To minimize this impact, RTPtv uses a restart interval of 6 MCUs. The use of 6 MCUs retains many of the same benefits as 3 MCUs without as large an increase in bitrate. In addition, the larger MCUs allows RTPtv to be used at a wider range of bitrates without needing to adjust the MCU size for better results. One question that is left unanswered by this report is how the cost of using few, but larger, network packets compare to other methods that use smaller, and more numerous, packets. In other words, is the difference in network impact trivial or significant?

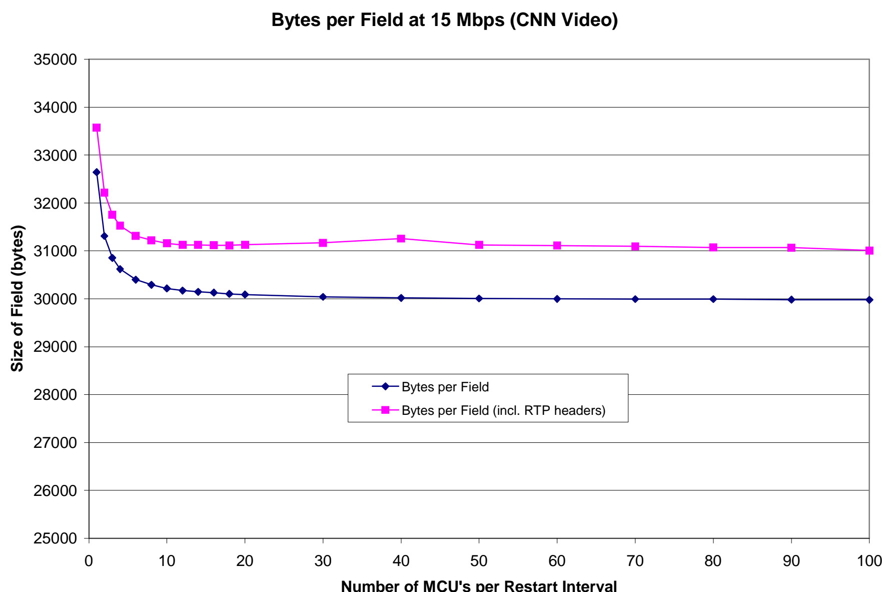


Figure 15: Average bitrate of the video stream, as a function of the number of MCUs in each Restart Interval.

4.2.2 Scheduling Playback of Frames on Video Client

By default, the RTPtv video client attempts to schedule playback so that each video frame appears on the screen exactly one second after it was captured on the server. The one second delay is an arbitrary goal selected because it negates factors such as network delay and jitter without causing an excessive amount of buffering on the client. Although a one second delay is too large for interactive conferences, it is suitable for conventional webcasts where the viewer has almost no means of knowing that such a delay exists.

The delay allows buffering to mask problems caused by a client not running NTP and allows more data to be stored at the OS level, thus mitigating the impacts of the OS CPU scheduler on the client applications. For example, a client video application without buffering must be executed roughly 30 times per second (once for each video frame). However, the OS driver allows for 4 frames to be buffered, and thus the client application can be run fewer times per second since the application can send multiple frames to the driver at a time (assuming that there is room in the driver).

Having multiple frames buffered at the OS level also helps with synchronization. In order to achieve accurate synchronization, the video client must know how many frames are buffered in the OS driver. This information is necessary because the amount of buffering will affect when the next frame output by the client will be displayed. To address this problem, the driver was enhanced so that an `ioctl()` call can check how many frames are buffered in the OS driver. This information is useful since you can

assume that each buffered frame will add a 1/29.97 second delay, but it does not provide a completely accurate value because the next frame to be output could be dequeued anywhere between 0 and 1/29.97 seconds from the current time.

The use of a hardware device for video playback does help in providing accurate video playback. It has been shown by Claypool and Tanner that jitter in video frame playback has a substantial impact on the perceived quality [Claypool and Tanner, 1999]. This jitter is unrelated to the jitter present within a network. It is caused by how the OS CPU scheduler decides when an application can, or cannot, execute. When a separate hardware device is used for video playback, the impact of the CPU scheduler does not affect the hardware device assuming that sufficient buffering can be maintained in the driver.

The one-second delay is not always desirable, and thus RTPtv provides an "interactive" mode where the delay is reduced to less than 250ms—the intended minimal delay is 187.5 ms, but the actual delay could be larger due to network delay.

The video client uses several buffers at the application level to store video frames; each video buffer stores a single frame (two fields for D1, one for CIF). At regular intervals a video client must decide which video frame to copy to the OS driver—these decision points occur when `select()` reports that a video buffer is now available in the driver. Each application level buffer can be in one or more of the following states:

- *available*: buffer is empty
- *checked-out*: buffer is currently being used to receive a video frame from the network
- *queued*: buffer contains a whole and complete video frame that is ready to be displayed
- *previous*: buffer contains the most recent video frame that was received from the network
- *last*: buffer contains the video frame that was most recently copied to the OS buffer

Figure 16 shows the transition state model. When a video buffer is empty it is considered *available*. When the first network packet for a new video frame is received a video buffer is reserved to receive the data, causing it to transition from *available* to *checked-out*. Once the video frame has been fully received, it is considered *queued*, and after being copied to the LML33 it is once again marked as *available*. A buffer is considered *previous* if it is the most recent frame to have been received from the network. The *previous* flag is used so that the JPEG loss recovery mechanism knows which video buffer it should use to get data to substitute for a lost packet. The *last* flag is primarily used when there are zero *queued* frames (which typically occurs due to the absence of data in the RTP stream). This flag is used to mark which video frame should be output in this type of a situation. In other words, if a video client has exhausted its supply of buffered frames and no new frames have been received from the network then the client will continue to display the last frame that it received. There is one, and only one, buffer that has its *previous* flag set (the same is true for the *last* flag).One final note is that the *previous* and *last* flags are also important for ensuring that an *available* video buffer is not reused while it is in either of these two states.

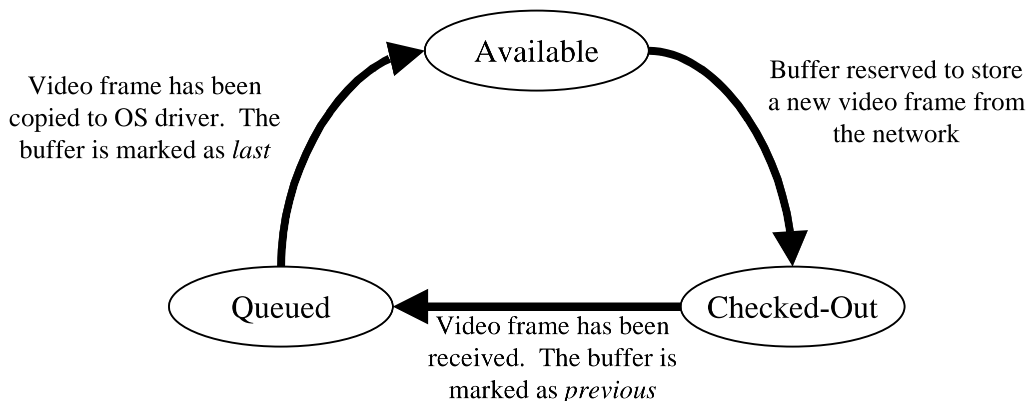


Figure 16: Transition State Model for Video Buffers.

As mentioned above, the *previous* flag is used to perform loss recovery. Under heavy loss conditions, such as in a network that can not handle multiple-megabit multicast streams, a client may never receive a whole video frame. To resolve this issue, the video client is designed to load a JPEG file into a buffer and mark this buffer as *previous*. Although this JPEG file image (currently a solid-blue background) will not match the video that is being received, it can be argued that a partial video image is better than none (plus the file image will eventually be completely covered by data from the RTP stream).

The use of the video file only works for situations where the JPEG stream uses restart intervals of the same size. If the sizes do not match, the video client will first wait until a whole and complete video frame has been received. A possible avenue for addressing this problem is to make available to the video client a representative restart interval for each possible size, and then use the appropriately-sized restart interval for the RTP stream. Another technique would involve using data from the network packet itself to substitute for missing data. RTPtv does not attempt to modify the restart interval size, so the only time that a mismatch should occur is if an RTPtv client receives a stream from a different RTP server application.

One note about the recovery code is that the use of substitute image data is sometimes made apparent in the fixed-bitrate mode. This is primarily because a wide array of quantization tables are used in the fixed bitrate mode, and the substitute data may have been taken from an image that used a different quantization table. As a result, the region of video that was substituted from another source may appear to be brighter, darker, or slightly different in color from the rest of the image. The visual effect is very similar to those described in the section on the RTP JPEG Q field, and they are most visible in situations where the quantization table scaling factor is changing rapidly, such as when the video is fading to black (e.g., the black screen that is used to transition between commercials). At this point in time a solution to the problem has not been devised, although the problem is not considered significant. If the variable bitrate mode is used, this problem does not occur because each JPEG image uses the same quantization table.

When it is time for the video client to send a video frame to the OS driver, it examines the video buffer at the front of the queue and converts that buffer's RTP media timestamp to an NTP timestamp (if no frames exists in the queue then the *last* frame is used without regard to its timestamp). The timestamp conversion involves two steps. The first step uses information from the RTCP sender reports to convert the buffer's RTP media timestamp to an NTP timestamp. The NTP timestamp is defined in terms of the server clock, so a correction for clock-skew is performed to adjust the NTP timestamp to be relative to the client clock. This clock-skew adjustment step (which can be disabled by the user) serves to adjust for clocks that differ (due to a lack of NTP running on both machines), but it also serves to eliminate the impact of network delay on scheduling the video buffer playback. For example, even if both the server and client are using NTP, the client will perceive a small (less than one second) skew in the clocks—even if a multi-second delay was encountered by the network then the client would still be able to schedule

using what, to it, appears (after clock-skew adjustment) to be a one-second delay (although this is assuming that the RTCP and RTP streams are delayed by an equal amount in the network, which we feel is safe to assume).

The video client has a desired *target range* for when the next video frame should have been captured. This target range is calculated by taking the current system time, subtracting the desired amount of delay (e.g., one second), and making an adjustment to account for the number of video frames already buffered in the OS driver. The target range is an interval that starts at that point in time, and goes back 1.667 frame times into the past (relative to the start of the interval). If the video buffer's NTP timestamp is found to be too "old" relative to this target range, the frame is discarded. If the video buffer is found to be too new (or all of the remaining video buffers were discarded for being too old), the "last" video frame is used.

An interval width of 1.667 frames (or 55.6ms) is actually a much larger interval than would otherwise be desired, but in using RTPtv we have found the value to provide acceptable results. The width can be thought of as a generous 1-frame-time, and is primarily needed because the LML33 board can not playback video at the full 29.97 frames/second.

The LML33 board captures video at 29.97 frames per second but it can only play back video at 29.856 frames per second. In other words, the server process sends frames at a rate that is 0.114 frames per second faster than what the client can handle. Consequently, the video client must occasionally drop a frame (on average every 8.772 seconds)—this frame drop occurs as part of the frame selection process mentioned in the preceding paragraphs, and no additional logic was needed to handle this unexpected behavior of the LML33 board.

When using an interval width larger than 1.667, we found the constraints to be too loose (causing inaccuracies in frame selection), while for smaller values the interval was too restrictive (e.g., a frame could get discarded for being too old, only to be used during the next frame selection process as a *last* video frame because the subsequent video frame is considered to be too new).

5 Audio Client/Server

This section describes some of the more interesting aspects of the audio client and audio server components of RTPtv. The audio server will first be described first, followed by the audio client.

5.1 Audio Server

The RTPtv audio server can transmit audio using one of the following codecs: *PCM u-law* [PCM, 1988], *Linear 8* (L8), and *Linear 16* (L16). The audio server can transmit at any sampling rate supported by the sound card, but it uses 16,000 samples/second, by default. The server uses stereo audio by default, but it also supports monaural audio. The techniques that are used to handle the loss of audio network packets are the most interesting aspect of the audio components. These techniques are discussed in the following section.

5.2 Recovery of Lost Audio Packets

A common problem that must be faced by network audio applications is to cope with the loss of data in the network. At higher audio sampling rates (e.g., 16,000 or higher), or with small packet sizes, it is usually possible to insert silenced audio as a substitute for a lost network packet. The use of silence audio can sometimes create a noticeable distortion, especially if preceding or succeeding audio packets were also lost. Experimentation with RTPtv has shown that when an audio packet is lost the immediately preceding and/or succeeding audio packets are almost always received.

To handle this type of loss behavior, support for *redundant audio* (RED) [Perkins, et al. 1997] was implemented in RTPtv. The use of RED allows a lost packet to be reconstructed using redundant information stored in the succeeding packet of the audio stream. As shown below in Figure 17, an audio

client can handle the loss of a network packet by using information stored in the next packet in the stream. Although this approach does not completely handle the situation when consecutive packets are lost, such occurrences are rare (especially over Internet2, although the use of RED over the "commodity Internet" has been reported in [Hardman, et al., 1995]).

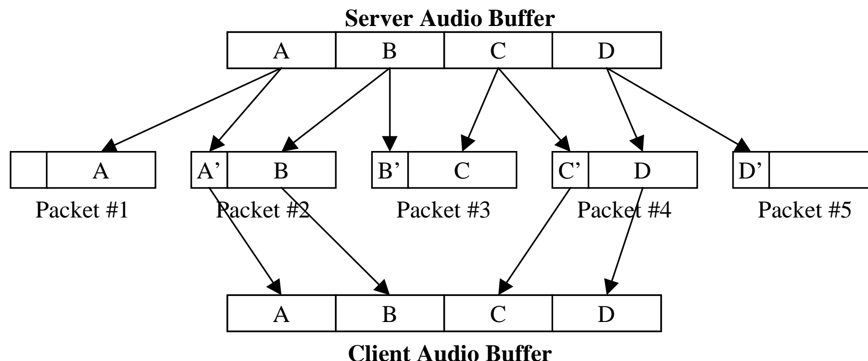


Figure 17: Recovery of a Lost Audio Packet (#3) using Redundant Audio.

To conserve bandwidth, the redundant audio is typically encoded using a low-quality audio codec. In the case of RTPtv, the *primary audio* is transmitted using a stereo L16 representation, while the *redundant audio* is transmitted using monaural L8 audio. Although the redundant audio is one-quarter the size of the primary audio, it proves to be of sufficient quality to substitute for lost data: the L8 audio is converted to L16 and the monaural audio is converted to stereo. The redundant audio could be reduced to one-eighth the bandwidth of the primary audio by using 8,000 samples/second instead of 16,000 samples/second, but the RED specification requires that both audio encodings be represented using the same sampling rate. This requirement is too stringent, and should perhaps be relaxed to require that the redundant audio be sampled at a rate that is an integer factor of the primary audio's sampling rate. Although few hardware devices support multiple sampling rates at the same time, it would be relatively trivial for an application to up-convert the redundant audio so that it is the same sampling rate as the primary audio.

Figure 18 shows the overall organization of an RTP audio packet that uses RED. By default, each RTPtv audio packet contains 291 samples of audio. When using redundant audio, 291 samples is the most that will fit within a 1,500 byte packet. The standard sampling rate for RTPtv is 16,000 samples/second, which means that each packet contains 18.1875ms worth of audio. The RTP specification suggests that each packet store 20ms of audio, so although RTPtv is using higher quality codecs the increase in packet rate is actual quite negligible (although the size of each network packet is considerably larger when compared to the 160-byte packets that are used for PCM u-law audio at a sampling rate of 8,000).

RTPtv primarily uses the stereo L16 codec. This codec is an uncompressed representation of audio, and was selected for several reasons. First, the codec is supported by almost all sound cards. Second, L16 is one of the few high quality audio codecs that is supported by other RTP implementations. Third, the RED codec is one of the few audio codecs that can handle data loss and is supported by other RTP implementations (unlike some of the other loss recovery techniques mentioned in [Perkins and Hodson, 1998]). The RED codec is typically used with only a select number of audio codecs, and L16 is one of them. Lastly, the L16 codec can be used for a variety of sampling rates (e.g., "CD quality sound") and audio channel quantities (e.g., "surround sound", although RTPtv does not currently support this particular feature).

Other audio techniques, such as "silence suppression," were not considered because these techniques tend to degrade audio quality (e.g., clip certain sounds). In addition, it is customary for silent audio (i.e., "dead air") to never be present in a television broadcast, and thus silence suppression would perhaps

never get invoked. The MP3 audio codec is more efficient than L16 in terms of bandwidth, but this saving comes at the expense of slightly degraded audio quality, increased CPU requirements for encoding, and fewer options for data loss recovery.

A sampling rate of 16,000 was selected because it seemed to be the best choice among the sampling rates that are commonly supported by both sound cards and software applications. For those situations where improved audio fidelity is required (e.g., for a music concert or MTV), the sampling rate can be increased.

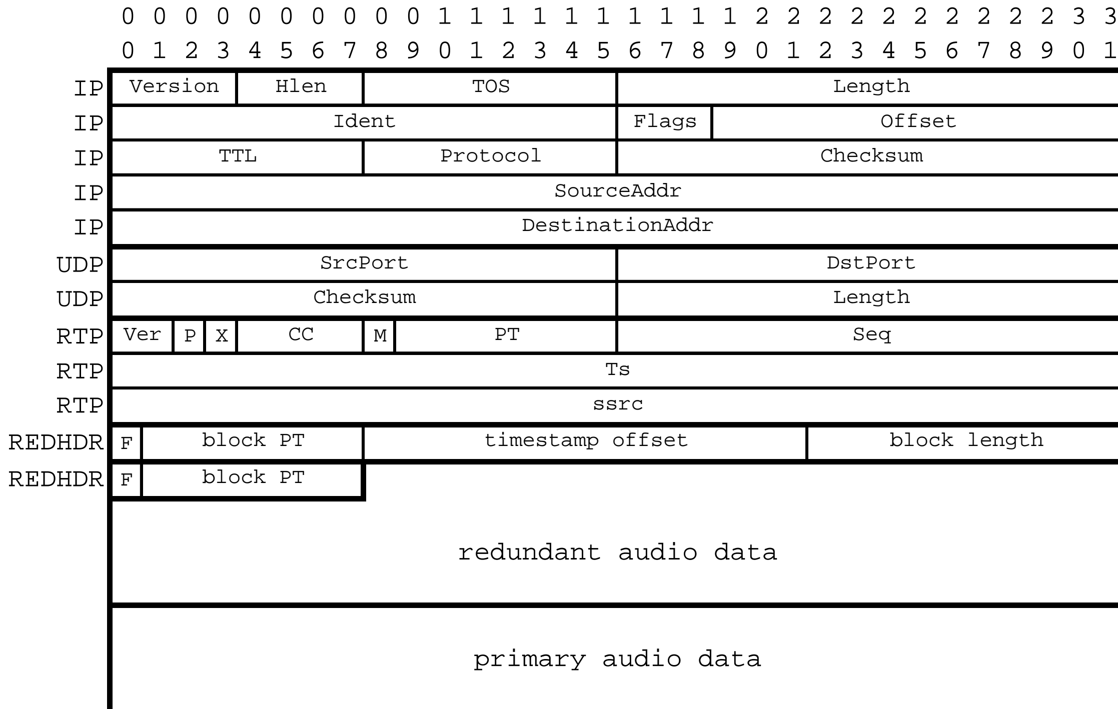


Figure 18: Example depiction of RTP Headers and Payload Data in each audio network packet.

5.3 Audio Client

The RTPtv audio client can receive all of the types of audio that can be transmitted by the RTPtv audio server. Although the RTP specification defines identification numbers for many audio codecs, others codecs were intentionally left undefined with the intent that the identification numbers can be arbitrarily defined for each media session. As a result, the RTPtv audio client was designed so that the user can specify/redefine the media identification numbers that represent each codec.

The transmission of audio using RTP may, at first, seem like a trivial matter. In fact, tools such as *vat* were implemented in the straightforward manner that ignores the clocking inaccuracies of a sound card. Before describing the details of the *vat* implementation, we first review the information provided by the RTP header. The RTP header contains a number of fields, only a few of which are essential for transmitting audio. These essential fields are: 1) the media type contained in the packet, 2) a packet serialization number that can be used to detect lost packets, and 3) a timestamp marking when the data in the packet was captured. In the case of audio, the media timestamp is defined to increase at a rate equal to the sampling rate of the audio stream. For example, a PCM u-law audio stream has a sampling rate of 8,000, so the timestamp counter increases by 8,000 every second.

Since the RTP timestamp is defined in terms of the sampling rate of audio, many RTP implementations use the sampling rate of their sound card to clock audio transmissions. For example, if the timestamp starts at 0, the timestamp for each packet can simply be updated by incrementing the

counter by the number of audio samples that have already been captured and transmitted. Unfortunately, this approach is not sufficient for high-quality applications. Due to differences among sound cards, such as variations in chips, oscillator crystals, and temperature, each sound card will vary in terms of how accurately it captures or plays back audio. Consequently, each audio server will capture and transmit audio at a rate that is slightly faster or slower than the sampling rate used by a client sound card.

The heterogeneity in sampling rates is typically quite small, and thus an audio stream may need to transmit continuously, or have long talk spurts, in order for problems to be apparent to the listener. If the sender is broadcasting at a rate faster than the client sound card, then the OS audio buffer will eventually overflow and audio will be silently dropped when the next call to `write()` fails. Conversely, if the sender is transmitting at a slower rate, short disruptions in the audio can be heard when the OS playback buffer is emptied before the next incoming block of audio is received. The differences in sampling rate are almost never an issue for audio applications that only transmit audio during a talk spurt, since the silence period between talk spurts gives slow audio cards a chance to empty their buffers, while fast sound cards will run out of audio to play and restart playback once the next talk spurt is scheduled to begin.

A final implementation consideration for RTP audio implementations deals with applications that rely on participants in a multicast session to have synchronized clocks. For many RTP audio implementations, the timestamps in the RTP stream will be clocked by the sound card, while the reports in the RTCP stream will be clocked by the computer's system clock. Because two clocks are used in the transmission, a skew can develop over time between the RTP and RTCP streams.

A more complete and accurate solution is used for RTPtv. Rather than using two clocks for an audio transmission, a single clock is used (in particular, the system clock). The use of a single clock serves to solve and/or eliminate many implementation issues that are pertinent to a single machine, including the synchronization between the RTP and RTCP streams. Because the audio card is producing and/or consuming audio at a rate that is unequal to the system clock, some special steps are taken in software to handle the differing clock rates in the sound card and system clock.

The overall process involves monitoring the rate at which audio samples are produced or consumed by the sound card—the audio server is interested in monitoring the number of bytes that have been produced, while the client is concerned with the number consumed. For ease of discussion we will only consider issues that pertain to an audio client, keeping in mind that an analogous process is carried out by the RTPtv audio server.

If the sampling rate of the sound cards is not monitored, synchronization with video begins to noticeably degrade in as few as 10 to 15 minutes. Synchronization can completely disappear in 30 to 45 minutes. Table 3 shows how many bytes per second are produced or consumed for several sound cards using stereo L16 audio at a 16,000 sampling rate. A perfect sound card would capture and playback exactly 64,000 bytes per second (i.e., 16,000*16 bits*2 audio channels). The SB Live card, considered by many to be a high-quality card, actually has the poorest accuracy of all the cards analyzed.

Table 3: Average bytes/second processed for Sound cards using 16-bit stereo audio at 16,000 Samples per Second.

	read	write
SB Live	64009.67	63994.04
ES1371	64001.16	64001.27
Gravis PnP Pro	64004.95	64004.98
Yamaha OPL3	64001.84	64001.82

The amount of skew caused by the differences in sound card sampling rates can be quite surprising. If one hour of audio is captured by the "Gravis PnP Pro," 60*60*64004.95, or 230,417,820 bytes of data will be read. During that same time period, an "SB Live" card would play 60*60*63994.04, or 230,378,544 bytes of data. At the end of this hour, the amount of data captured would exceed the amount

played back by 39,276 bytes, resulting in a skew of 0.614 seconds between these two sound cards. If this same experiment was performed by capturing data from the SB Live card and playing it back via another SB Live card, a skew of 0.879 seconds would result after one hour. This amount of skew may seem small, but it is enough to prevent precise synchronization from being maintained. A design goal for RTPtv is to have 24 hours of continuous playback (if not an indefinite playback duration) with no loss of synchronization. As a result, a skew of no more than -15 to 30ms can be present for NTSC [Steinmetz and Engler, 1993].

When the audio card playback process is started, the system time is recorded by the audio client. By convention, to play sound an audio application copies data to the driver as a group of audio samples called a *fragment*. After the RTPtv audio client has copied a fragment to the sound card, it uses the system clock to determine exactly how many total bytes should have been consumed by the sound card from the time that the sound card playback process was started. This estimated value is compared against the actual number of bytes that have been consumed by the sound card. If the sound card has consumed too few samples, a few bytes are deleted. Conversely, if too many samples have been produced then a few bytes are inserted. To avoid creating audible artifacts, at most 2 samples are inserted or deleted at a given time.

There are two methods for determining how many samples the sound card has actually consumed. The first method is to count how many samples have been copied to the driver by the audio client. The second method involves polling the audio driver to see exactly how many bytes have been consumed by the sound card. There are obvious tradeoffs between the two approaches. The first method is available regardless of the audio API, but the results are less accurate (if not inaccurate) and makes it more difficult to make gradual adjustments. The results are less accurate because decisions are not based on the actual number of bytes processed by the sound card. Instead, the figure also includes the number of bytes stored in the audio driver buffer. Some APIs allow an application to ask how many bytes are stored in the driver, but many drivers are inaccurate (sometimes producing wrong results), or the numbers are approximated to the nearest fragment size (instead of reporting the exact number of bytes).

The second method provides a more accurate value, but this capability is not provided by all audio APIs. The OSS API can only provide this type of functionality if the driver supports real-time capability, but few drivers support this feature [OSS, 2000]. The *ALSA* driver and API provides this functionality, and also supports other features (such as obtaining the exact time that the driver started the sound card) [ALSA 2001]. The current implementation of RTPtv only attempts to make adjustments for sound card inaccuracies on systems that support the ALSA API—the OSS API lacks certain features, and those features that would be useful are often implemented in a unique fashion for each driver.

The ALSA API allows an application to query how many total bytes have been produced by a sound card. Although this API provides a virtually real-time result, it does not report a time associated with the sample count. To address this problem, the software polls the system time immediately before and after the audio API call. The two time values then serve as high and low watermarks for the API call. If the byte count exceeds the high watermark, a few samples (up to 2) are deleted. If the byte count is below the low watermark, up to 2 samples of silence are inserted.

Compared to other operating systems, UNIX-based systems offer highly precise system clocks. Because of this precision, fine-tuned adjustments can be made to correct for the inaccuracies of the sampling rate of the sound card and it is uncommon for there to be a need to adjust for more than 2 samples at a time. In fact, RTPtv is able to maintain an accuracy range of 24 bytes. The software keeps track of a sound card tendency (either fast, slow, or "no recent change"), and how many bytes have already been corrected for by the software.

If the sound card is found to be diverging outside of the expected range, and its diverging in a direction that is consistent with its past tendency, an adjustment is made. For example, if the software has already corrected for a skew of 100 bytes, and it now detects a skew of 104 bytes, then a correction for the 4 bytes will be performed. However, if the skew has reduced to 96 bytes or 90 bytes then no adjustment is made—the assumption is that a small amount of CPU scheduling delay may have caused a slight miscalculation in the byte adjustments, and the expectation is that the error from the miscalculation

will not be present during the next check and/or the sound card will have continued to increase its skew back towards 100.

Sound cards rarely change their fast/slow tendency, but it does occur. The only example that has been witnessed so far is that a sound card may initially appear to be running fast, only to later reveal a tendency to be slow—it is likely that some initialization issue in the drivers may cause this behavior. The RTPtv software checks to see if a sound card has changed its tendency. If the sound card skew is 24 or more bytes in the opposite direction of its tendency, and this has occurred twice in a row, the sound card tendency is assumed to have changed and the software adapts.

The counters and other related state for sound card monitoring can be susceptible to a counter underflowing or overflowing. As a result, some actions are performed to "rebalance" the state after certain counters exceed certain specified values. Without this re-balancing, certain positive values could overflow and become negative, or vice versa, wreaking havoc on the system.

The act of adding and removing audio bytes is a crude, but effective, technique. Although a more thorough solution would involve extrapolation of the audio data, the process of inserting/removing audio samples is effective and does not result in any noticeable audio distortions. What would be most helpful for network audio applications is for an audio API to report when each audio fragment was sampled. Presently there is no audio information associated with an audio stream, and thus an application must make inferences.

Another area where inferences might need to be made is in the area of buffer underflow and overflow. Buffer overflow is rarely a problem for a server (unless that server is heavily loaded), but audio underflow can have a large impact on a client (especially for synchronization). If underflow occurs on a client, the client must re-index into its stream to account for the fact that some of the data was not played at the proper time (most often because the data was not received from the network and had to be recovered).

Unfortunately, most audio APIs (such as OSS) do not have a mechanism for determining if a buffer underflowed. To address this issue, the audio client monitors the number of fragments that are buffered in the OS driver and assumes that underflow has occurred (or will occur) if there are fewer than two audio fragment buffers in the driver. This buffer occupancy check is performed just before the next call to `write()`—the assumptions made by RTPtv are conservative, yet there is still the possibility that underflow could still occur (e.g., a context switch could occur between the time the buffer occupancy is checked and the call to `write()` is made).

In contrast, the ALSA API makes it very easy to determine if underflow or overflow has occurred. An application call to `read()` or `write()` will fail with a special status code when a buffer has overflowed or underflowed, respectively. This approach to buffer management reduces the number of system calls and guarantees that an application will always know if a buffer problem caused the capture or playback process to be disrupted.

6 Synchronization Revisited

The preceding sections have outlined the steps taken by each of the RTPtv processes to unify their clocks to ensure that each of the media streams remain synchronized. As mentioned earlier, these measures solve the clocking issues pertinent to a single machine, but they do not directly address the differences between the clocks of two different computers. The difference in clock rate is not an issue for short-term sessions, and for a long-term session the NTP protocol can be used to synchronize the clocks of the session members. For RTPtv the main importance of NTP is that it can regulate the rate at which clocks advance; the fact that NTP keeps clocks in-step with a global clock is a side benefit.

Each server process transmits media streams with accurate timestamps that remain coordinated with their companion RTCP streams. As a result, each client process is able to obtain and maintain synchronization using the "classic" technique that was presented: buffer each stream, index into those streams at corresponding points in time, and begin playback. This presents the issue of where exactly should each client index into a media stream. By default, RTPtv tries to buffer and playback data in such

a way that the media streams are played back at a time that is one second after the time they were captured. This step involves converting the RTP media timestamps to NTP timestamps (using information from the RTCP stream), and also using the RTCP stream information to adjust the NTP timestamps to account for the clock skew (if any) that exists between the server and client.

The clock skew technique also helps in another situation. RTPtv supports an optional "interactive mode," whereby the client attempts to achieve a 0.1875 second delay, rather than 1.0 second. If the network delay is quite high, this low-latency goal would be impossible to achieve. However, if the clocks on the server and client are synchronized, the clock skew calculation will result in a skew that equals the network delay. As a result, this value is used to adjust the target 0.1875 second delay to account for the network delay (i.e., more network delay results in a larger total latency).

This technique differs from that used by the *vat* and modified *vic* pair developed at the UCL. Rather than using a direct process-to-process IPC mechanism, the RTPtv clients remain synchronized because they have been designed to delay based on the local clock, rather than the state of another RTPtv client. This technique can be thought of as using a level of indirection: rather than processes talking directly with each other, they coordinate indirectly via the system clock. The task of obtaining the system clock is quite accurate and does not involve any context-switching issues as with the "direct IPC" approach.

One final note is that the RTPtv clients do not just synchronize and blindly playback. Instead, they constantly monitor the hardware and make local adjustments, as necessary, to account for the inaccuracies of the hardware.

7 Experimental Results / Experiences

The following sections describe some of the experiences that were gained through the use and development of RTPtv. These include results regarding bitrates, round-trip time in Internet2, the quality of synchronization in broadcast TV, the visual quality of RTPtv at certain bitrates, CPU requirements of RTPtv, and the amount of code that was required to implement the system.

7.1 Network Experimentation using RTPtv

When RTPtv is configured to use the fixed-quality mode for video, a surprising variation in bitrate was observed. Depending on what video images are being captured, the bitrate was observed to vary from as little as 3 Mbps (3,059,235.2 bits/second) to as much as 13 Mbps (13,049,145.6 bits/second). A variation in bitrate occurs even when the fixed-bitrate mode is used. An experiment was performed to determine the magnitude of the variation, and the results are shown in Table 4 (Table 5 shows results for the audio codecs). The values in Table 4 are not necessarily the absolute bounds of what can be experienced. Rather, they provide an accurate guideline for what can be expected. The target bitrate is presumably lower than the actual bitrate since the target value does not include the overhead of the network packets or other factors such as restart markers.

Table 4: Actual Target Rate compared to Actual Behavior.

Target Rate		Actual Packets/sec.		Actual Bits/sec.	
bits/field	bits/second	Minimum	Maximum	Minimum	Maximum
40,000	2,397,600	238.4	242.4	2,592,758.4	2,597,198.4
80,000	4,795,200	473.8	481.8	5,017,569.6	5,073,144.0
160,000	9,590,400	901.2	946.6	9,860,550.4	9,951,660.8
240,000	14,385,600	1,380.8	1,427.2	14,735,284.8	14,824,065.6
320,000	19,180,800	1,832.0	1,993.8	19,586,841.6	19,650,812.8
400,000	23,976,000	2,356.8	2,557.4	24,346,091.2	24,576,584.0
480,000	28,771,200	2,845.6	3,150.2	29,307,225.3	29,512,086.0
variable	N/A	289.0	1,220.0	3,059,235.2	13,049,145.6

Table 5: Actual rate of packets/second and bits/second for various audio codecs.

Codec	Sampling Rate	Codec Bitrate	Packets/second
PCM-u	8,000	64,000	27.49
RED: L8 mono	8,000	128,000	27.49
L8 mono	16,000	128,000	54.98
RED: L8 mono	16,000	256,000	54.98
L8 stereo	16,000	256,000	54.98
RED: L8 stereo	16,000	384,000	54.98
L16 mono	8,000	128,000	27.49
RED: L16 mono	8,000	192,000	27.49
L16 stereo	8,000	256,000	27.49
RED: L16 stereo	8,000	320,000	27.49
L16 mono	16,000	256,000	54.98
RED: L16 mono	16,000	384,000	54.98
L16 stereo	16,000	512,000	54.98
RED: L16 stereo	16,000	640,000	54.98
L16 stereo	44,100	1,411,200	151.55
RED: L16 stereo	44,100	1,764,000	151.55

The RTPtv streams encounter very little packet loss within Internet2, and when a packet is lost it is more than likely that the loss occurred within the local network of the server and/or client, rather than in the long-haul network paths. In fact, it has been common for local viewers to incur higher packet loss than remote Internet2 viewers due to the characteristics of the local network at UC Berkeley. When loss does occur, the video stream generally encounters a greater percentage of packet loss than the audio stream. This difference may be due to routers that use queuing techniques that place a higher percentage of loss on higher bandwidth streams, or may instead be due to the fact that when a video frame is transmitted it causes several network packets to be transmitted in rapid succession.

During the early development stages of RTPtv an experiment was performed to see if packet loss could be reduced by scheduling packets to be transmitted at a particular time (rather than transmitting all in short period of time). These experiments showed that the UNIX timers are too course to allow packet scheduling to work effectively at the application level. Consequently, the act of "packaging" each network packet individually, rather than as a bulk set that it processed and transmitted as a group, seems to be the only reasonable way for high bandwidth applications to space out the amount of time between packet transmissions.

One situation that can occur is for network packets in a stream to become reordered due to a routing change in a network. Although such events can occur, we have yet to experience any reordering while using RTPtv. The software is designed to detect and react to reordered packets, but at the present time RTPtv discards reordered packets rather than attempt to use them. This is because RTPtv is designed to react to network loss as soon as it is detected, and if a "missing" packet were to appear later in a stream then RTPtv would need to "rollback" its state to utilize the data. Support for these types of situations is complex, and therefore unwarranted given how rarely a packet reordering will occur (especially given the fact that there are very few network paths in Internet2).

7.2 Roundtrip Time within Internet2

One of the features of RTP is that a server can use RTCP to estimate the roundtrip time (RTT) to a particular client. Although this information is not used by the RTPtv software, it does provide interesting data about packet behavior across the United States using Internet2. Figure 19 shows that the RTT can vary from one moment to another, without any discernable pattern in the behavior. The inability to see a

pattern may be due to the fact that RTCP reports are sent at five second intervals, which may be too infrequent to provide this type of result. Figure 20 depicts the same information as Figure 19, except that the RTT results have been sorted. Figure 20 shows that most of the data points are below 0.07 seconds, but if this value were used in order to minimize the network latency then roughly one-quarter of the packets would arrive too late to be of use.

Since humans have a limited ability to detect latency within a stream, it was decided that RTPtv should be designed to operate at a latency below the threshold of human perception, but not so aggressive as to force packets to be thrown away because they arrived too late.

Another curious behavior that was observed is that multicast traffic appears to have a larger round-trip time than unicast traffic. Due to possible inaccuracies in the RTT feature of RTP, it may be unwarranted to make concrete conclusions about this behavior, but during one experiment between UC Berkeley and Purdue it was observed that the unicast traffic had an RTT of 80ms, while the multicast traffic took 90-100ms.

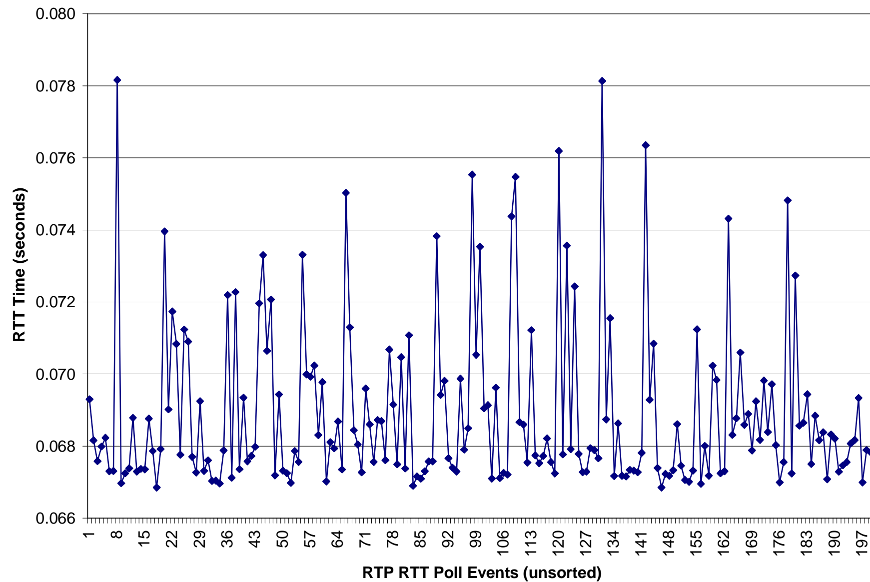


Figure 19: RTCP Roundtrip time from UC Berkeley to Michigan via Internet2.

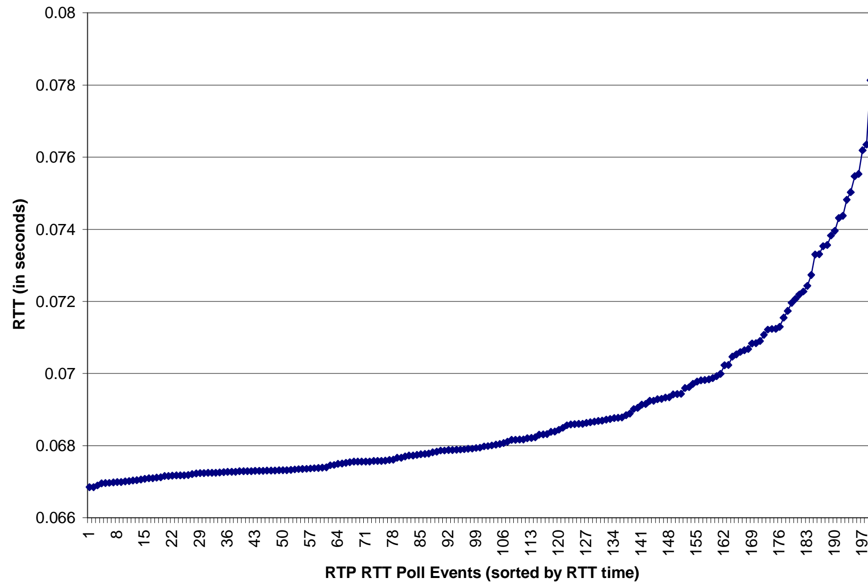


Figure 20: RTCP Roundtrip time from UC Berkeley to Michigan via Internet2 (sorted).

Within the local networks at UC Berkeley, a striking difference was found in packet loss rate between streams that used unicast versus those that used multicast over the identical network links. For example, during one test a 10 megabit unicast video stream encountered 0.5-0.8% packet loss. When that stream was restarted using multicast, the loss rate increased to 82.9%. Another curious aspect of this behavior is that the multicast problem was unidirectional: trading the roles of client and server (thus reversing the direction of the data flow) resulted in a multicast stream that encountered packet loss rates of only a few percent.

7.3 Synchronization in Broadcast Television

As part of the process for "fine tuning" the synchronization it was necessary to closely scrutinize broadcast television to look for synchronization errors. Surprisingly, it is not a rare occurrence to find a broadcast that is using unsynchronized audio and video. These synchronization errors are typically quite small, and would likely be unnoticed by the average "couch potato," but they do exist. For example, these errors can be prevalent for reporters who are broadcasting from a remote location, but they can also occur for shows that have been edited in a sloppy fashion.

7.4 Quality of Video at Certain Bitrates

One topic that has been ignored up until this point is the perceived quality of the video at various bitrates. The perceived quality for a given bitrate is largely dependent on the characteristics of the video images, but the following is a general characterization of the quality of certain bitrates for D1 video (for CIF, roughly one-half the amount of data is required):

- **2.5 megabit video:** too low in quality to be tolerable; the video is almost exclusively a mosaic image of large blocks, and often lacks color.
- **5 megabit video:** decent quality, although blocks and other JPEG visual artifacts can be seen upon close inspection.

- **10 megabit video:** good quality, but it is still possible for the trained eye to find small visual artifacts for certain types of video
- **20 megabit video:** at this level the quality is essentially identical to the source video; higher bitrates can product a slightly crisper picture but are not usually worth the added bandwidth.

7.5 Use of RTPtv for a Seminar Broadcast

RTPtv was used to broadcast the Berkeley MIG Seminar on May 2, 2001 across Internet2 at 15Mbps [Rowe, 2001]. Up until that time, the seminar was broad using RealPlayer (250Kbps) and the Mbone Tools (500-700Kbps). The feedback from the RTPtv broadcast of the seminar was quite encouraging, with several viewers requesting that better analog audio and video equipment be used in the studio/classroom for the broadcast. One viewer also complained about the "room lighting." These comments show a clear change in the "quality bottleneck" of the system that has moved from the transport mechanism (i.e., the network), to the actual analog audio and video.

The seminar broadcast was not without flaws. A transient problem prevented a second video stream from being broadcast, and it was discovered that the LML33 board does not interact well with video routing switches that are not frame-accurate. Both of these problems have been addressed: the first problem disappeared, while the second issue was addressed by having the video server restart the LML33 board when it detects that video frames are no longer being captured (a better long-term solution involves using a frame-accurate video switch for changing between video sources).

7.6 RTPtv CPU Resource Consumption

Each RTPtv application uses less than 5% of a 600Mhz Intel Pentium III processor. In fact, a combination of two video servers and a single audio server can operate reasonably well on a 200Mhz Intel Pentium Pro processor. The amount of CPU resources used by a server is largely dependent on the bitrate that is being produced by the server. The RTPtv audio server and client processes use an extremely small amount of CPU resources. The RTPtv video client can have the widest variation in CPU requirements. This is because, during a period of high loss, the data recovery code can consume the majority of the CPU time that is used by the video client (upwards of 90% of the time). As a result, the amount of CPU resources can vary based on the amount of network data loss that is encountered.

At UC Berkeley several instances of RTPtv servers are run simultaneously on the same machine. Typically no more than two video servers and one audio server are used simultaneously, but we have used three LML33 boards in a single machine and the driver does support up to four LML33 boards. Given past experiences with the LML33 board, it is anticipated that, in a system with a fast CPU, the PCI bus will become saturated with traffic before the CPU is fully utilized. As a result, it seems more cost effective to distribute a collection of LML33 boards across several old and inexpensive machines rather than to place several (more than three) LML33 boards within a smaller collection of fast, more expensive machines.

7.7 Lines of Code in RTPtv

The complete implementation of RTPtv requires nearly 20,000 lines C++ of code. Table 6 list how many lines of code were devoted to the various modules of the system:

Table 6: Classification of lines of C++ Code (including comments) in RTPtv.

Category	Lines	Subtotal
Audio		5,069
Audio Client	541	
Audio Server	699	
Audio Buffering & Conversion	2,042	
Audio Stream Processing / Mgmt.	1,787	
Video		7,339
Video Client	391	
Video Server	898	
Video Buffering & Conversion	2,466	
Video Stream Processing / Mgmt.	3,584	
Miscellaneous		7,149
Networking / Sockets	1,287	
GUI Code / Event Handlers	1,687	
RTCP Processing / Tracking	1,645	
RTSP Support	559	
General RTP State Mgmt.	1,971	
Total		19,557

The source code is available at http://bmrc.berkeley.edu/~delco/rtptv/.

Given the large amount of code that was written, it is reasonable to ask whether it would have been better to implement RTPtv using a general multimedia toolkit, such as the *Open Mash* Multimedia Toolkit from UC Berkeley [OpenMash, 2001]. These toolkits have good support for many RTP codecs, but they currently do not support media synchronization or higher quality audio codecs. Also, they only partially support certain video codecs. During the development of RTPtv certain JPEG processing features were ported to Open Mash, such as updates to the parsing of RTP headers and support for JPEG loss recovery, so that RTPtv streams can be viewed by *vic* and other Open Mash application.

8 Future Work

There are several additional features that could be added to RTPtv. Some of those features include the ability to record and playback television streams (either on-demand, or at preset times). This functionality can already be achieved through the use of other RTP applications such as RTP tools [RTPtools, 2001] or MARS [Schuett, et. al., 1999], but an internalized capability (ideally using RTSP) would be much easier to use. Support for additional audio codecs would be useful, and the RTPtv code is modularized in a fashion that would facilitation their inclusion.

Another aspect that can be pursued further is to revise the RTP JPEG standard to better support other types of MJPEG streams, such high-bitrate streams and streams that use a larger variety of quantization tables.

Eventually the cost of high-quality hardware MPEG implementations will decrease, at which point it will be beneficial to adjust the video management software to handle an MPEG codec.

9 Conclusion

This report has presented an RTP-based system for transmitting high quality, synchronized, audio and video streams across IP-based networks. Obtaining highly precise synchronization over a non-trivial amount of time requires careful management of every phase of the end-to-end process. This report has provided an overview of several of the techniques and mechanisms that are used to achieve and maintain synchronization. In addition, the report has described the algorithms and techniques used for handling and concealing data loss.

Acknowledgements

This research was partially funded by NSF Internet Technologies Program Grant 9907994 and NSF Instrumentation Equipment Grant 9512332. I would like to thank Bob Riddle for his work in helping to test and promote RTPtv. Denis DeLaRoca and Greg Cook have also been invaluable in helping to test and use the system. I would also like to thank my OpenMash colleagues, particularly Paul Huang and Lloyd Lim, for their help in modifying and enhancing OpenMash to support the RTP streams produced by RTPtv.

References

[Apple, 2001] Apple Computer, Inc. QuickTime Player version 5. Software available online at http://www.quicktime.tv/. May 2001.

[ALSA, 2001] Advanced Linux Sound Architecture Project. ALSA audio driver and API version 0.5.11. Software available online at http://www.alsa-project.org/. May 2001.

[Bacher, et. al., 1996] D. Bacher, A. Swan, and L. A. Rowe. rtpmon: a third-party RTCP monitor. Proceedings of the Fourth ACM International Conference on Multimedia, 1996. Version 1.0a7 of the software is available at ftp://mm-ftp.cs.berkeley.edu/pub/rtpmon/.

[Berc et al., 1998] L. Berc, W. Fenner, R. Frederick, S. McCanne, and P. Stewart. RTP Payload Format for JPEG-compressed Video, Request for Comments (Proposed Standard) RFC2435, Internet Engineering Task Force, October 1998.

[Carpenter, 1996] B. Carpenter, Architectural Principles of the Internet, Request for Comments (Informational) RFC 1958, Internet Engineering Task Force, June 1996.

[Cerf and Kahn, 1974] V. G. Cerf and R. E. Kahn, A protocol for packet network interconnection, IEEE Trans. Comm. Tech., vol. COM-22, V 5, pp. 627-641, May 1974.

[Clark and Tennenhouse, 1990] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In SIGCOMM, pages 200-208, Philadelphia, Pennsylvania, Sept 1990. ACM.

[Claypool and Tanner, 1999] M. Claypool, and J. Tanner. The Effects of Jitter on the Perceptual Quality of Video. In Proceedings of ACM Multimedia, Orlando, Florida, November 1999.

[Deering and Cheriton, 1990] S. E. Deering, and D. R. Cheriton. Multicast routing in datagram networks and extended LANs. ACM Transactions on Computer Systems. Vol. 8, Issue 2. 1990.

[Deering, et. al., 1996] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C. Liu, and L. Wei. The PIM Architecture for wide-area multicast routing. IEEE/ACM Transactions on Networking. Vol. 4, Issue 2. April 1996.

[Estrin, et al., 1998] D. Estrin, D. Farinacci, A. Helmy, D. Thaler, S. Deering, M. Handley, V. Jacobson, C. Liu, P. Sharma, L. Wei. Protocol Independent Multicast-Sparse Mode (PIM-SM): Protocol Specification (Experimental) RFC 2362, Internet Engineering Task Force, June 1998.

[Fenner, 1997] W. Fenner. Internet Group Management Protocol, Version 2 (Proposed Standard) RFC 2236, Internet Engineering Task Force, November 1997.

[Handley and Jacobson, 1998] M. Handley, V. Jacobson, SDP: Session Description Protocol (Proposed Standard) RFC 2327, Internet Engineering Task Force, April 1998.

[Handley et al., 2000] M. Handley, C. Perkins, E. Whelan, Session Announcement Protocol (Experimental) RFC 2974, Internet Engineering Task Force, October 2000.

[Hardman, et al., 1995] V. Hardman, M. Sasse, M. Handley and A. Watson; Reliable Audio for Use over the Internet; Proceedings INET'95, Honolulu, Oahu, Hawaii, September 1995.

[ISI, 1981] Information Sciences Institute, Internet Protocol. Request for Comments (Standard) RFC 791, Internet Engineering Task Force, March 1992.

[ISO, 1991] ISO DIS 10918-1. Digital Compression and Coding of Continuous-tone Still Images (JPEG), CCITT Recommendation T.81. 1991.

[ISO, 1993] ISO/IEC International Standard 11172; Coding of moving pictures and associated audio for digital storage media up to about 1.5Mbits/s, November 1993.

[ISO, 1994] ISO/IEC International Standard 13818; Generic coding of moving pictures and associated audio information, November 1994.

[Jacobson and McCanne, 1992] V. Jacobson and S. McCanne. The LBL audio tool vat. Available from ftp://ftp.ee.lbl.gov/conferencing/vat/, July 1992.

[Kouvelas, 1998] I. Kouvelas. A Combined Network, System, and User Based Approach to Improving the Quality of Multicast Audio. Ph.D. thesis, University College London, London, England, May 1998.

[LML33] Product description page for the LML33. Available online at http://www.linuxmedialabs.com/lml33doc.html.

[McCanne and Jacobson, 1995] S. McCanne and V. Jacobson. "vic : A Flexible Framework for Packet Video". In Proceedings of ACM Multimedia'95, pages 511-522, San Francisco, CA, November 5-9, 1995.

[Microsoft, 2001] Microsoft Corporation. Windows Media Player version 7. Software available online at http://www. microsoft.com/windows/windowsmedia/. 2001.

[Mills, 1992] D. Mills. Network Time Protocol (version 3) specification, implementation and analysis. Request for Comments (Proposed Standard) RFC 1305, Internet Engineering Task Force, March 1992.

[Hodson and Perkins, 1999] Orion Hodson and Colin Perkins. Robust audio tool (RAT) version 4. Software available online at http://www-mice.cs.ucl.ac.uk/multimedia/software/rat-4.0 December 1999.

[OpenMash, 2001] OpenMash version 5.1.4. Software available online at http://www.openmash.org/. May 2001.

[OSS, 2000] Open Sound System Programmer's Guide 1.11. Available online at http://www.opensound.com/pguide/oss.pdf. November 7, 2000.

[PCM, 1988] International Telecommunication Union. ITU-T Recommendation G.711. Pulse code modulation (PCM) of voice frequencies. November 1988.

[Perkins, et al. 1997] C. Perkins, I. Kouvelas, O. Hodson, V. Hardman, M. Handley, J.C. Bolot, A. Vega-Garcia, and S. Fosse-Parisis. RTP Payload for Redundant Audio Data (Proposed Standard) RFC 2198, Internet Engineering Task Force, September 1997.

[Perkins and Hodson, 1998] C. Perkins, O. Hodson,. Options for Repair of Streaming Media (Informational) RFC 2354, Internet Engineering Task Force, June 1998.

[Peterson and Davie, 2000] L. Peterson and B. Davie. Computer Networks: A Systems Approach. Morgan Kaufmann Publishers. San Francisco, CA. 2000.

[RealNetworks, 2001] RealNetworks, Inc. RealPlayer version 8. Software available online at http://www.real.com. 2001.

[Rowe, 2001] L. Rowe, Berkeley Multimedia, Interfaces, and Graphics Seminar, http://bmrc.berkeley.edu/courseware/mig, 2001.

[RTPtools, 2001] H. Schulzrinne, and P. Pan. RTP Tools version 1.17. Software available online at http://www.cs.columbia.edu/IRT/software/rtptools/. April 2001.

[Schuett, et. al., 1999] A. Schuett, R. Katz, and S. McCanne. A Distributed Recording System for High Quality Mbone Archives. Proceedings of the First International Workshop on Networked Group Communication. Pisa, Italy. November 1999.

[Schulzrinne, 1996] H. Schulzrinne, RTP Profile for Audio and Video Conferences with Minimal Control, Request for Comments (Proposed Standard) RFC1890, Internet Engineering Task Force, January 1996. This specification is currently being updated, and the current "work in progress" is available as draft-ietf-avt-profile-new-XX.txt.

[Schulzrinne et al., 1996] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. Request for Comments (Proposed Standard) RFC1889, Internet Engineering Task Force, January 1996. This specification is currently being updated, and the current "work in progress" is available as draft-ietf-avt-rtp-new-XX.txt.

[Schulzrinne et al., 1998] H. Schulzrinne, A. Rao, and R. Lanphier. Real Time Streaming Protocol (RTSP). Request for Comments (Proposed Standard) RFC2326, Internet Engineering Task Force, April 1998.

[Steinmetz and Engler, 1993] R. Steinmetz, and C. Engler. Human Perception of Media Synchronization. Technical Report, No. 43.9310, IBM European Networking Center, Heidelberg, Germany. August 1993.

[TclTk, 2001] Foundry for Tcl and Tk. Software available online at http://sourceforge.net/foundry/tcl-foundry/. Version 8.3.3 was released April 2001.

[Wallace, 1991] G. Wallace, The JPEG Sill Picture Compression Standard, Communications of the ACM, April 1991, Vol 34, No. 1, pp. 31-44.

[Zoran, 1997] Instant Replay (H33) Reference Design: PCI-based Consumer Video Editing Board. Design overview available online at http://www.zoran.com/products/pcvideo/pcvideord.asp. Design specification available online via ftp://ftp.zoran.com/. May 1997.

RPC API Appendix

This section describes the RPC API that is available for use by Tcl/Tk scripts. BNF grammar is used in the following descriptions. Each RPC call returns a result, and each command description provides a description of the possible results that may be returned. In addition, a command may also return one of the following generic error messages: *syntax error*, *invalid state*, or *connection error*.

`rtv_alloc`

Allocates a state object. Returns *rtvXX* (where XX represents a two digit number) or *no available state*. For example, the first time *rtv_alloc* is executed a result of *rtv00* should be returned.

`rtv_set rtvXX (attribute value)+`

Changes one or more attributes to a different value in the state object *rtvXX*. A list of defined attributes is provided in the next section. If the service is currently active, then the changes are propagated to the active service. Returns *changed ("{true}" | "{false}")+*, where the value represents if a particular value change was successful. For example, if a service is active then a call to *rtv_set rtv00 bitrate 160000 showlogo true* should receive a response of *changed {true} {true}*.

`rtv_get rtvXX local (attribute)+`

Returns values of one or more specified attributes in local state object *rtvXX*. Returns *values ('{'value'}')+*. A value will be *ERROR* if lookup error encountered for a particular attribute. For example, a call to *rtv_get rtv00 local showlogo kk bitrate* might get a reply of *values {false} {ERROR} {160000}*.

`rtv_get rtvXX remote (attribute)+`

Returns values of the one or more specified attributes for the active service that corresponds to *rtvXX*. The values are obtained from the source server, and the results are stored in the *rtvXX* state object. See previous paragraph for return syntax.

`rtv_status hostname port (audio|video) (server|client) deviceNumber`

Polls the specified RPC management process at hostname/port to get the status of the specified device. Returns whatever response it gets from the server.

`rtv_start rtvXX hostname port (audio|video) (server|client) deviceNumber ...`

Requests that the specified RPC management process at hostname/port start a service using the specified device. Returns *started* or *not started*. Valid command options include:

`rtv_start rtvXX hostname port "audio" "server" deviceNumber "multicast" address port port`

Requests that the specified RPC management process at hostname/port start a dual unicast/multicast service (first port is multicast, second port is unicast) using the specified device. Returns *started* or *not started*.

`rtv_start rtvXX hostname port "audio" "client" deviceNumber "unicast" address port`

Requests that the specified RPC management process at hostname/port start a client that connects to the specified server hostname and port using the specified device. Returns *started* or *not started*.

rtv_start rtvXX hostname port "audio" "client" deviceNumber "multicast" address port

Requests that the specified RPC management process at hostname/port start a client that connects to the specified multicast session address and port using the specified device. Returns *started* or *not started*.

rtv_stop rtvXX

Requests that the active service corresponding to *rtvXX* should attempt to terminate itself. Should return either a connection error, or the phrase *stopped*.

rtv_kill hostname port (audio|video) (server|client) deviceNumber

Requests that the RPC management process at hostname/port kill the active service corresponding to the specified device. Returns *killed* or *not killed*. For example, you could make a call to *rtv_kill 127.0.0.1 4000 video server 0*.

rtv_dealloc rtvXX

Deallocates the state corresponding to *rtvXX*. If there is an active service that corresponds to this state, then it is NOT terminated. Returns *bad request* or *deallocated*.

rtv_ping "host" hostname port

Attempts an RPC ping to the specified hostname and port. Returns *pong* on success, or a generic error message.

rtv_ping rtvXX (app|host)

Attempts an RPC ping to the host's RPC service, or the application service that is currently associated with the *rtvXX* state object. Returns *pong* on success, or a generic error message.

RPC Attributes and Values

Generic Attributes	Attribute Type	Default Value	Comments
hostaddress	string	127.0.0.1	Address of Server
hostrpcport	integer	4000	TCP port of RPC process on server
apprpcport	integer	5000	TCP port of active service (if active)
mediatype	set	video	Type of service (video or audio)
servicemode	set	server	Mode of service (server or client)
devicenumber	integer	0	Hardware device number to use
multicastaddr	string	<empty string>	Address of multicast session
multicastport	integer	0	Port of multicast session
multicastttl	integer	64	TTL to be used for multicast session
unicastaddr	string	<empty string>	Address of source to be used by client
unicastport	integer	0	Port of source to be used by client
alwaysxmit	boolean	false	Specifies if server should always multicast
lowdelay	boolean	false	Specifies if low-latency should be used
monitorskew	boolean	true	Used to ignore change in network skew
serviceactive	boolean	false	Specifies if the current service is active

Video Attributes	Attribute Type	Default Value	Comments
videosize	set	d1	Specifies video size (d1 or cif)
bitrate	integer	80000	For server, specifies number of bits per field
logomode	boolean	true	For server, used to disable use of logo screen
showlogo	boolean	false	For server, used to show/hide logo
useextqtable	boolean	true	For server, disables RTP extension header
framerate	integer	0 (full framerate)	For server, specifies a custom framerate
fixedq	boolean	false	For server, specifies if variable bitrate is used
showoverlay	boolean	false	For client, enables display of video on monitor

Audio Attributes	Attribute Type	Default Value	Comments
samplingrate	integer	16000	Species the sampling rate of the audio stream
mixeradj	boolean	true	Used to disable adjustments to audio mixer
usered	boolean	true	For server, used to disable use of RED
usestereo	boolean	true	For server, used to specify mono or stereo audio
audioformat	set	l16	For server, specifies l16, l8, or pcmu audio
syncaudio	boolean	true	For client, used to disable synchronization

RTP Media Types	Attribute Type	Default Value	Comments
mono8pcmu	integer	0	RTP Media Type of PCM u-law audio
red	integer	125	RTP Media Type of RED codec
mono16	integer	124	RTP Media Type of monaural Linear 16
stereo16	integer	126	RTP Media Type of stereo Linear 16
mono8	integer	127	RTP Media Type of monaural Linear 8
stereo8	integer	128	RTP Media Type of stereo Linear 8