Adaptive Control for Packet Video

Soumen Chakrabarti* Randolph Wang†

Computer Science Division, University of California, Berkeley, CA 94720

soumen@cs.berkeley.edu, rywang@cs.berkeley.edu

Abstract

We describe a portable and robust software mechanism for adaptive frame rate control for real time packet video transfer and viewing in workstation environments. No special hardware or system support is assumed. Our contributions include (1) a responsive feedback system for dynamic presentation layer rate control in response to network load, (2) a simple and effective transport layer flow control scheme built upon an unreliable network protocol, and (3) data structures and scheduling support for integrating the rate and flow control mechanisms. We demonstrate that excellent jitter control is achieved by gracefully trading image resolution, and loss rates can be drastically reduced by appropriate flow control.

Keywords: Packet video, adaptive compression.

1 Introduction

Motion video transmission for real time viewing over packet switched networks is an established area of multimedia research [4], with attractive applications such as video conferencing and picture telephony. However, it has performance requirements on transmission delay, jitter, loss, available bandwidth and process scheduling priority that conventional networks and operating systems have, until recently, been unable to provide with guarantee. This results in at least two problems for a packet video application. First, maintaining a smooth frame rate under variable network conditions becomes a considerable challenge. Inability to detect and adapt to a constantly changing environment leads to jittery frame rates. Second, the nature of packet video transmission requires careful flow control techniques. Flow control schemes that are too conservative under-utilize network resources while transmissions that are too aggressive lead to high packet losses.

Recently, the Tenet group at Berkeley have explored specialized network protocols and operating system support for such real time applications [3]. It might be argued that the increasing bandwidth brought about by advances of network technology, together with better operating system and network software support will obviate the need for applications that adjust to varying network availability. However, such optimism may not be completely justified.

- Guaranteeing part of a shared resource encourages underutilization. Adaptive applications, on the other hand, can modify their behavior to fully take advantage of the existing resources. As the available bandwidth increases, there will be larger number of users and more demanding applications; hence the need for adaptive applications is likely to remain and such "intelligent" applications are better equipped to fully utilize the resources.
- While real time network and operating system support is likely to deliver more strict guarantees, it also comes at a considerable cost. Complicated changes will have to be made to operating systems on *all* participating machines (including routers). Furthermore, applications that run in such specialized environments may not run at all, or run poorly, in less hospitable environments. An adaptive application, on the other hand, can deliver satisfactory performance in most cases without special support. It is thus more portable.

In this paper we explore our implementation of high quality packet video. We contribute a successful end-to-end software solution entirely at user level, using the following techniques.

- To achieve a smooth frame rate at the presentation layer, we introduce the notion of *load lines*, which are used to dynamically adjust the compression factor according to the network load.
- By making the sender a slave of the receiver, we implement a transport layer flow control that reduces packet loss while maximizing throughput.
- To integrate the rate and flow control mechanisms above, we build data structures and scheduling support for unreliable transmission and reordered packets.

The software solution enables rapid prototyping and evaluation of many design choices using *actual* images, machines and networks, unlike off-line simulations of coders and networks as in much of the literature.

In §2 we introduce the compression scheme and show, using both analysis and empirical evidence, that bandwidth requirement can be controlled in a

*Supported in part by a U.C. Regents Fellowship, and the Advanced Research Projects Agency of the Department of Defense monitored by the Office of Naval Research under contract DABT63-92-C-0026. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

†Supported in part by Digital Equipment Corporation (Systems Research Center and External Research Program), and the AT&T Foundation.

In the 1994 IEEE International Conference on Multimedia Computing and Systems, Boston.

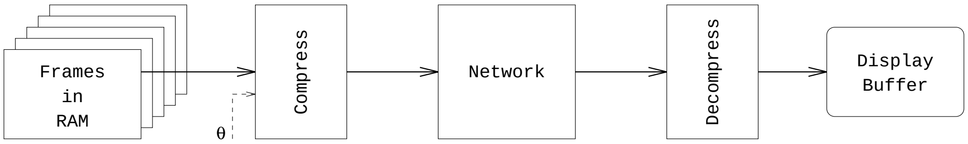

Figure 1: The basic modules for compression, transfer and decompression.

predictable way using one *threshold* parameter. §3 describes the design of the feedback control system, exploring the mechanisms for rate and flow control, and describing the integrated software design. §4 describes the performance of our prototype. Extensions and future work are suggested in §5. Concluding remarks are made in §6.

2 Compression Mechanism

Our compression scheme is described in this section. We make estimates of the compression factor and introduce the notion of a *load line*, which is central to our adaptation scheme described later on.

Sending uncompressed motion video at reasonable size and frame rate puts a heavy burden on the network. A large number of signal processing algorithms [6] are known for encoding a video image into fewer bits, reducing the *spatial redundancy* and hence bandwidth requirement. The spatial redundancy arises out of the fact that for most common images, surfaces and light sources are piecewise continuous. In the case of motion video, there is also *temporal redundancy* — frames close together in time are often highly correlated.

2.1 Quadtree Encoding

For our purpose, we use *quadtree compression*, a lossy spatial compression scheme. Consider, for simplicity an $n \times n$ block of pixels on a grey-level image, where a pixel is a numeric value representing brightness. Divide the square into four $\frac{n}{2} \times \frac{n}{2}$ squares, and compute the sum S_{11}, S_{12}, S_{21}, S_{22} of the pixels in each of the four squares. Compute their average $\overline{S} = \big(S_{11} + S_{12} + S_{21} + S_{22}\big)/4$. If for all $i, j \in \{1, 2\}$, $|\overline{S} - S_{ij}| < \theta$, where θ is a prescribed threshold, then the whole $n \times n$ block is represented by the average pixel value, $4\overline{S}/n^2$. If not, we recursively apply the algorithm on the four $\frac{n}{2} \times \frac{n}{2}$ squares. Thus the subdivision of the image space is logically a quadtree structure.

Quadtree encoding is not new [5], but the criterion for subdividing a block is often based on the variance of all pixel values in the block. We use a different criterion for two important reasons. First, the effect of θ on the bandwidth requirement is well understood in our method (as we shall see in the next section). This enables us to build the control protocol more systematically. Second, only integer addition and shift

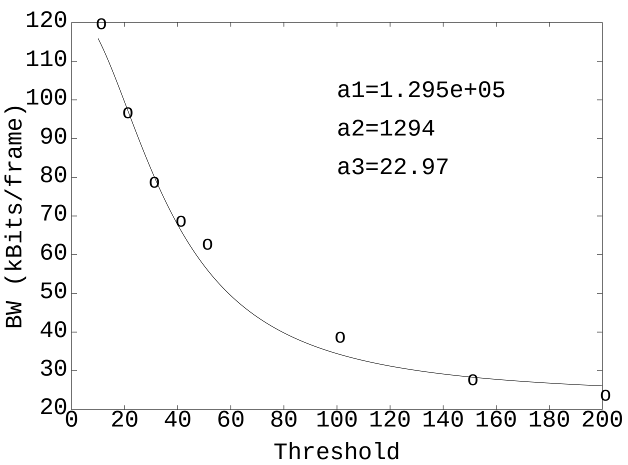

Figure 2: Experimental data from a typical image (`football`) shows close agreement with the analysis for a synthetic Gaussian noise image. The curve fitted is $G(\theta) = a_1/(\theta^2 + a_2) + a_3$, where G is the bandwidth requirement per frame when the value of the threshold is θ. a_1, a_2 and a_3 are curve fitting parameters explained in the text.

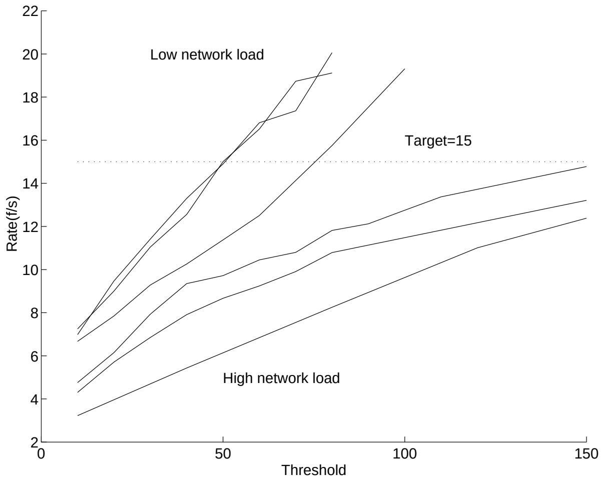

Figure 3: $F(\theta)$ has been plotted against θ at various fixed network load conditions, generating a family of curves, each called a *load line*. "Network load" is in abstract units — it is not necessary for calibrating the control mechanism. θ is also in arbitrary but significant units. The experiments were performed on a cluster of DEC 5000's on a 10 Mbits/s Ethernet.

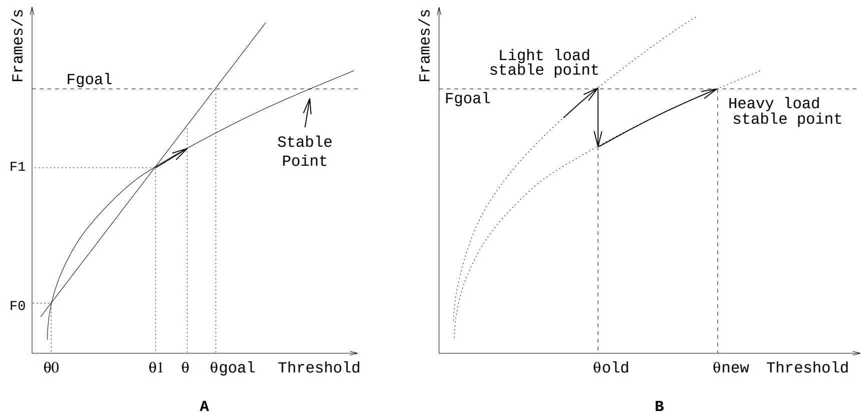

Figure 4: The adaptive threshold selection mechanism. Figure A shows how a stable operating point is arrived at under fixed network load. Figure B shows how the system reacts to a sudden increase in network load: the frame rate drops suddenly and the feedback control adjusts the threshold from θ_{old} to θ_{new}. Sudden reduction in network load is reacted to analogously.

are needed in our method, which are fast single cycle operations on most modern RISC microprocessors. In contrast, integer multiplication takes 11 cycles and division takes 36 cycles on a DEC 5000/133 with a 33 MHz MIPS R3000 microprocessor.

2.2 Adaptation

The thesis of our work is that compromising resolution in a controlled way is a superior alternative to suffering jittery frame rates, losing packets and/or skipping frames. In the quadtree encoder just described, the threshold is a means to decide what bandwidth an image needs for transmission. To motivate this, consider a (synthetic) Gaussian noise image — one where each pixel is an independent random variable obeying a Gaussian distribution with mean 0 (without loss of generality) and variance σ^2. Suppose that the threshold is θ and the full image is $n \times n$.

For each pixel with value X we thus have

$$\mathbf{Pr}\left[x < X < x + dx\right] \quad = \quad \frac{1}{\sigma\sqrt{2\pi}} e^{-x^2/2\sigma^2}\, dx. \quad (1)$$

We shall evaluate the largest subdivision b necessary such that, with very high probability, $b \times b$ tiles in the image need not be further divided (let $B = b^2$). Since the complete image is $n \times n$, and most $b \times b$ tiles are represented by a single pixel, n^2/b^2 is a good estimator of the number of pixel values we shall need to transmit. For a given value of b, we have, for $i, j \in \{1, 2\}$:

$$\mathbf{E}\left[S_{ij}\right] = 0 \qquad \mathbf{V}\left[S_{ij}\right] = \frac{B\sigma^2}{4}, \quad (2)$$

$$\mathbf{E}\left[\overline{S}\right] = 0 \qquad \mathbf{V}\left[\overline{S}\right] = \frac{B\sigma^2}{16}, \quad (3)$$

where S_{ij} and $\overline{S}$ are also normally distributed. The random variable of interest is $\overline{S} - S_{ij}$, which is also normally distributed with mean 0 and variance

$$\mathbf{V}\left[\overline{S} - S_{ij}\right] \quad = \quad \frac{5B\sigma^2}{16}. \quad (4)$$

For a normal distribution, it is known that

$$\mathbf{Pr}\left[\left|\overline{S} - S_{ij}\right| \le 3\sqrt{\frac{5B\sigma^2}{16}}\right] \quad > \quad 0.99, \quad (5)$$

so further block division is very likely to be prevented whenever θ is such that

$$3\sqrt{\frac{5B\sigma^2}{16}} \quad < \quad \theta, \quad (6)$$

$$\Leftrightarrow \qquad B \quad < \quad \frac{16\theta^2}{45\sigma^2}. \quad (7)$$

Thus we can expect, with very high probability, that the number of pixels G required to transmit a frame to be related to the threshold θ in the form $G = A/\theta^2$, where A is a constant depending only on the image (n and σ).

The above analysis is highly idealized. To make it more realistic, observe that situations where $\theta \to 0$ or $\theta \to \infty$ are modeled in an unrealistic fashion: near $\theta = 0$, we need to transmit no more than n^2 pixels, and even as $\theta \to \infty$, some basic transmission overhead has to be incurred. To build these extreme behaviors into our framework, we suggest the more realistic model

$$G(\theta) \quad = \quad \frac{a_1}{\theta^2 + a_2} + a_3. \quad (8)$$

In figure 2, we test our suggested model using a typical motion video scene (`football`). Excellent agreement is observed, with the fitting parameters being $a_1 = 129500$, $a_2 = 1294$, and $a_3 = 22.97$. From this we compute $G(0) = 122$ kBits/frame. The image

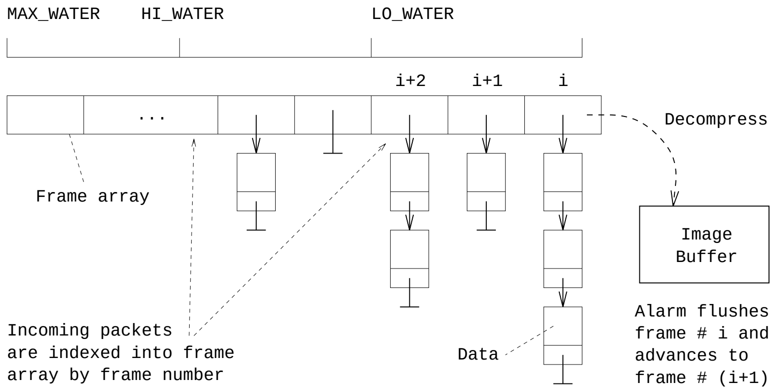

Figure 5: The leaky bucket mechanism for jitter reduction. It also provides more accurate flow control. Packets (possibly reordered) arrive and are enqueued into the frame array. The alarm clock goes off every 67 ms. The alarm interrupt handler decompresses all remaining packets for frame i, refreshes the display, and advances the current frame to $i+1$. The objective is to keep between `LO_WATER` and `HI_WATER` non-empty frames in the array.

was 160×120 pixels, each represented by one byte, which accounts for 153 kBits/frame. Considering that `football` is very different from Gaussian noise, this is a very good estimate. Also, $G(\infty) = a_3 = 22.97$ kBits/frame. This is because the coder did not start with the entire image — it tiled the image into 8×8 blocks before running the quadtree algorithm (so that image quality is not arbitrarily compromised). All real life images we have seen closely follow the synthetic analysis.

2.3 Load Lines

Having demonstrated, using a synthetic model as well as real life images, that transmission bandwidth is inversely related to threshold θ, we explore the implications of this observation. Consider a simple network model offering a constant bandwidth to the video communication. Since $G(\theta)$ is inversely related to θ, we would expect the frame rate F to be directly related to θ. Using the model in (8),

$$F(\theta) \quad = \quad a'_1 - \frac{1}{a'_2\theta^2 + a'_3}, \tag{9}$$

for appropriate constants a'_1, a'_2 and a'_3. Each curve in figure 3 is a plot of $F(\theta)$ against θ at a fixed network bandwidth (as far as experimental methods permit). The lower curves are for less available bandwidth. Varying available bandwidth was effected by controlled execution of other synthetic network workload with calibrated communication rate.

F should be a scaled reflection of G in the x-axis; for some curves, the resulting saturation (at high network load) is visible, while others (at lighter load) are fairly linear. A curve for a fixed network load is termed a *load line*. In the next section we show how we use the notion of load lines to stabilize the frame rate F.

3 The Control System

In this section we describe the design and implementation of the feedback control software. Figure 1 shows the basic modules for compression and transmission, as our starting point.

There are essentially two control knobs we can turn: the number of bits a frame is encoded in, and the number of bits that go through the network per unit time, to stabilize the number of frames transmitted (and shown) per unit time, and minimize packet losses while maximizing throughput. §3.1 describes the rate control mechanism and §3.2 describes the flow control mechanism.

3.1 Presentation Layer Rate Control

The analysis in §2 indicates that θ is a powerful means for controlling compression factor. Our goal is to set θ dynamically based on network "load", so that F is held constant:

- Find a usable, responsive and robust measure of the network load as it appears to the video application.
- Fix a stable operating point on the current load line. That is, choose a value of the threshold that, under current network conditions, gives the desired frame rate at maximum possible picture quality.
- When the load line shifts owing to changes in network load, recompute the threshold to settle at a new operating point with the same frame rate.

The central design idea is to use the load lines introduced in §2.3 to monitor recent trend in frame rates and threshold and thus select the next threshold value. Figure 4 illustrates how the load line is approximated by straight chords. The monitored variables of interest are the frame rate F and threshold θ, at the current time t, and at some recent past epoch $t - \Delta t$. Then we let θ_{goal} be

$$\theta[t] + \frac{\theta[t] - \theta[t-\Delta t]}{F[t] - F[t-\Delta t]}\Big(F_{\text{goal}} - F[t]\Big), \tag{10}$$

where F_{goal} is, for now, constant (15 in our case). This will change later when we integrate the solution with

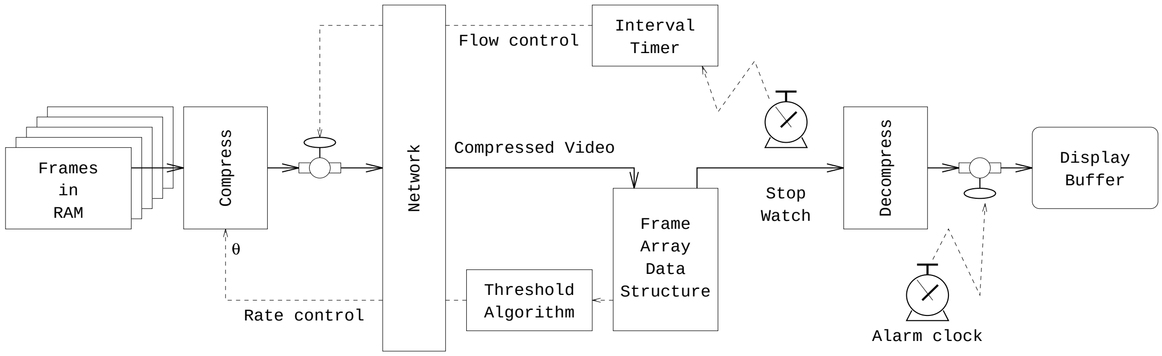

Figure 6: The complete system with rate and flow control.

flow control. We do not wish to be too bold in setting $\theta[t+\Delta t]$ to θ_{goal}, since the readings could be noisy. We dampen the response to favor stability using a linear back-off:

$$\theta' = \alpha\theta[t] + (1-\alpha)\theta_{\text{goal}}, \quad (11)$$

$$\theta[t+\Delta t] = \max\Big\{\theta_{\text{MIN}}, \min\{\theta_{\text{MAX}}, \theta'\}\Big\}, \quad (12)$$

where α is an empirically chosen smoothing constant. We used $\alpha = 0.5$, but the choice was not critical. When setting F and θ, we also clip them within some intervals $[F_{\text{MIN}}, F_{\text{MAX}}]$ and $[\theta_{\text{MIN}}, \theta_{\text{MAX}}]$ to ensure stability in extreme cases.

3.2 Transport Layer Flow Control

Even though `TCP` [1] provides a reliable and flow-controlled byte-stream, this protocol is too heavy-weight for our purpose. For real time video viewing, retransmission to make up for dropped packets is not practical. Furthermore, `TCP` handles flow control using dynamic window management, which as we shall see, is not suitable for our application.

We have chosen `UDP` as the transport layer. The price we pay for using the light-weight protocol is that we must provide customized flow control and mechanisms for dealing with out-of-order delivery. Consider the following first approximation to the sender and receiver code.

```
loop { /* SENDER */      loop { /* RECEIVER */
 compress a packet;       receive a packet;
 send a packet;           decompress a packet;
}                        }
```

Ideally, we would like the sender and receiver loops progress at approximately the same rate. Unfortunately, the code path taken by the receiver is longer than that of the sender. Without flow control, the receiver would be overrun by the sender and drop a large number of incoming packets.

In our prototype, we implement a simple but effective flow control to reduce receiver loss rates without compromising performance. The receiver simply measures the amount of time it takes to complete an iteration, using the stop-watch in figure 6. This loop time is conveniently piggy-backed onto the acknowledgement packets. Upon receiving the receiver cycle time, the sender compares that against the time elapsed between the transmissions of two successive packets. If the sender is running ahead, it puts itself to sleep until the two cycle times match.

Experiments suggest that this flow control scheme is more effective than window based schemes, under which a fast sender is allowed to let out bursts of packets which the receiver may not be able to cope with. Our flow control, on the other hand, provides instantaneous feedback to the sender and the control is at a finer grain.

3.3 Implementation

The threshold adjustment mechanism directly affects the compression algorithm, so it might be run at the sender. In this case, the sender's idea of the current frame rate (which is used to recalculate the threshold) is based on how fast it can inject packets into the network. Thus this is insensitive to

- End-to-end network load conditions.
- Receiver decompression and display activity.

The remedy is to realize that the specified *viewing rate* is what drives the system, and that is most reliably monitored at the *receiver*. Thus the threshold adaptation is moved to the receiver, and recomputed threshold values are transmitted back to the sender (forming a negative feedback path).

However, now the receiver will find that the sender is too slow to respond to its dictated threshold, resulting in *hunting* and load transients. We use a leaky bucket (figure 5) — incoming packets (possibly reordered) are indexed into a frame array with water marks `HI_WATER`, `LO_WATER` and `MAX_WATER`. The intention is to keep spare frames to display so that sender response delays are masked, and provide storage if the sender gets ahead.

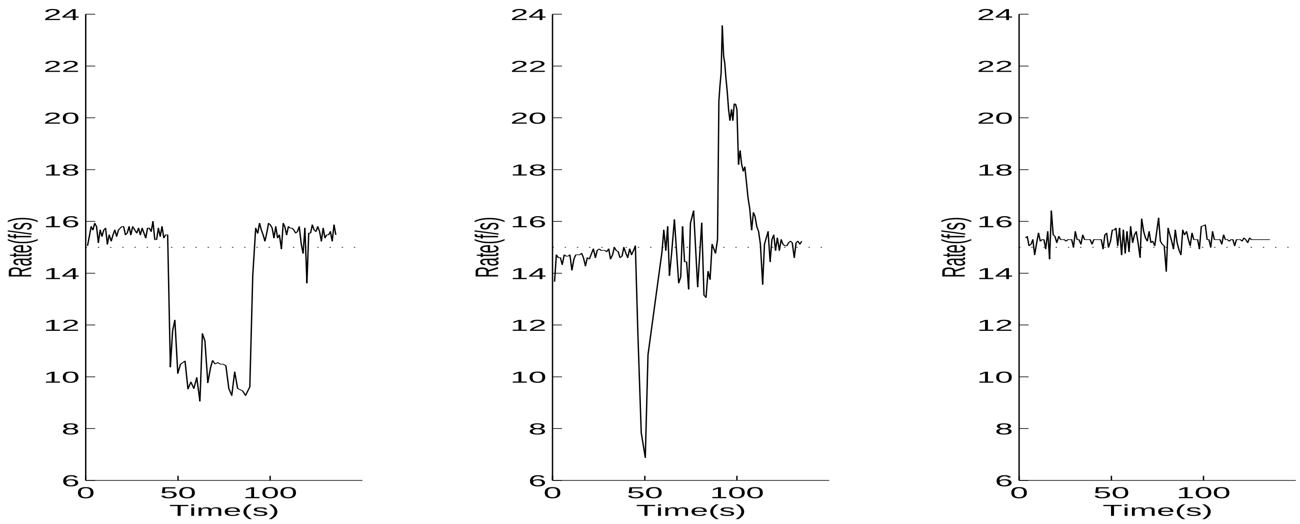

Figure 7: Steps in refining the compression and feedback control mechanism. Frame rate F has been traced in time t, with another process involved in data transfer between $t = 45$ and $t = 90$ seconds. The leftmost diagram shows a large dip in F during this interval. The middle diagram shows the stabilizing effect of our adaptive compressor, but because of the reaction delay, transients are still visible at leading and trailing edges. Adding a leaky bucket integrator smoothes down these fluctuations to give a very steady frame rate in the rightmost graph.

The basic system shown in figure 1 flushed the current frame to screen as soon as a packet bearing a greater frame number arrived (this was adequate for an Ethernet, not packet reordering networks). With frame arrays, display refresh at the receiver is not driven by packet arrival any more. Instead, an *alarm clock*, shown in figure 6, goes off every 1/15-th second. The alarm interrupt handler completes any further decompression needed from the current frame i and advances the current frame to $i+1$. With the frame array, our notion of F_{goal} changes. Since we are recomputing $F[t]$ and $\theta[t]$ quite regularly now (on interrupt), Δt is fairly constant. We thus approximate:

$$F'[t] = F[t] + \frac{\ell[t] - \ell[t-\Delta t]}{\Delta t} \qquad (13)$$

$$F'_{\text{goal}} = F_{\text{goal}} + \frac{\texttt{HI\_WATER} - \ell[t]}{\Delta t} \qquad (14)$$

where $\ell[t]$ is the 'water' level in the frame array, i.e., the number of non-empty frame lists. It is possible that $\ell[t] = \texttt{MAX\_WATER}$: in that case the sender is throttled down. Finally we set $F[t+\Delta t]$ and $\theta[t+\Delta t]$ as before. The foreground computation at the receiver consists of polling the frame array and making progress at decompression before the alarm goes off. When the alarm goes off, this ensures that few packets remain to be decompressed, so the image refresh occurs promptly.

The leaky bucket also provides for more accurate flow control. Without a small buffer, the receiver may not have packets to receive or decompress and consequently reports a false cycle time to the sender. Furthermore, the buffer space in the leaky bucket allows the receiver to schedule first the most urgent of the three events at any time — receiving packets, decompressing packets, and displaying frames, further smoothing frame rates and reducing packet losses.

4 Performance

We tested our design and implementation on a cluster of DEC 5000/133 workstations connected to a 10 Mbit/s Ethernet LAN. Later we also tested the system on a WAN. In qualitative terms, each of four researchers from the Tenet group (real time network researchers, familiar with usual standards of packet video quality) found that the improvement was remarkable. In this section we support these observations with concrete measurements.

Figure 7 shows the results of an experiment (on an Ethernet LAN) to determine how effective our jitter control method is. We ensured that there were essentially two ongoing network transactions: one for video transmission (which continues for the duration of the experiment) and a large data transfer from $t = 45$ through $t = 90$ seconds. Whereas frame rate drops to as low as 9 frames/s from the unloaded value of 16 frames/s, the stabilized system with our threshold adaptation holds the frame rate almost constant, close to the target rate of 15 frames/s.

Another improvement was in drastically reduced packet loss rates compared to versions without flow control. With a fixed threshold value and no flow control, as much as 29% packet loss is observed (see figure 8). High packet loss forces incomplete screen buffer refreshes, resulting in moving objects appearing to leave parts behind. Our flow control method reduces packet loss down to less that 3%, eliminating this problem.

The control mechanism was remarkably adaptive. We ran the software with all tuning parameters unchanged on a WAN, transmitting motion video from a workstation in the University of Wisconsin at Madison to a Berkeley workstation. Image quality was comparable with Ethernet performance.

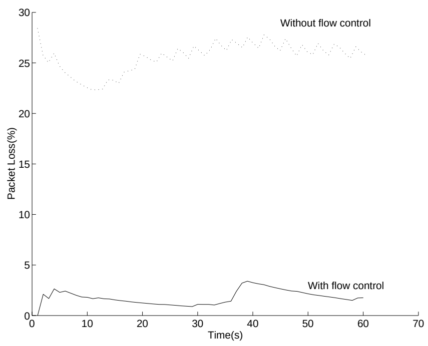

Figure 8: Our adaptive compression scheme reduces typical loss rates from 29% to 3%, eliminating premature screen refresh problems. A typical time trace is shown.

5 Extensions and Future Work

To contain the scope of the project, we have, in this paper, focused on engineering issues of video transmission over a packet switched network. The following may merit further investigation.

- Our control system has been empirically determined to be robust and stable. However, the analysis of bandwidth in §2 has been used only qualitatively in the design of the adaptation mechanism. The system is clearly non-linear, and delay in the feedback loop makes a formal stability analysis complicated, even though this may be a worthwhile exercise.
- The prototype is only for gray-scale images. There could be interesting engineering required in transmitting color images, because of higher bandwidth and synchronization requirements.
- The issue of audio transmission has not been addressed. Audio encoding, compression and transmission is fairly standard at this time, and adding on the functionality is essentially a programming effort, except that audio hardware is required [7], and the audio and video have to be kept synchronized [2].

6 Conclusions

We have described a software mechanism for adaptive threshold control for real time packet video transfer and viewing in workstation environments. A responsive, portable and robust feedback control has been designed to gracefully trade image resolution for frame rate stability. By appropriate flow control at the receiver, loss rates have been drastically reduced. Our prototype works equally well for Ethernet LANs and WANs. Finally, quantitative measurements of performance improvements have been presented.

Acknowledgements

Professor Domenico Ferrari suggested the project and some key ideas for monitoring and adjusting to network load. Most of the network and display code in figure 1 was inherited from Michael Gilge and Richardo Gusella at the International Computer Science Institute, Berkeley. Ramon Caceres, Mark Moran and Tom Fisher of the Tenet group, Tom Skibo (Urbana), Paul Haskell and Rick Hans (Berkeley) participated in many discussions. Chih-Po Wen helped with proof-reading.

References

[1] Douglas E. Comer. *Internetworking with TCP/IP*, volume 1. Prentice Hall, 1991.

[2] J. C. Cooper. Video-to-Audio Synchrony Monitoring and Correction. *SMPTE Journal*, 97(9):695–698, September 1988.

[3] Tom Fisher. Real Time Scheduling Support in ULTRIX–4.2 for Multimedia Communication. In *Proceedings of the Third International Workshop on Network and Operating System Support for Digital Audio and Video*, San Diego, November 1992.

[4] E. A. Fox. Advances in Interactive Digital Multimedia Systems. *Computer*, 24(10):9–21, October 1991.

[5] Michael Gilge and Riccardo Gusella. Motion Video Coding for Packet-Switching Networks — An Integrated Approach. In *SPIE Conference on Visual Communication and Image Processing*, Boston, November 1991.

[6] T. R. Hsing. New Directions in Video Coding and Transmission. *Proceedings of the SPIE - The International Society for Optical Engineering*, 974:74–81, August 1988.

[7] G. Maturi. Single Chip MPEG Audio Decoder. *IEEE Transactions on Consumer Electronics*, 38(3):348–356, August 1992.