A Distributed Real-Time MPEG Video Audio Player *

Shanwei Cen, Calton Pu, Richard Staehli,
Crispin Cowan and Jonathan Walpole

Department of Computer Science and Engineering
Oregon Graduate Institute of Science and Technology
Portland, Oregon, USA
{*scen, calton, staehli, crispin, walpole*}*@cse.ogi.edu*

Abstract. This paper presents the design, implementation and experimental analysis of a distributed, real-time MPEG video and audio player. The player is designed for use across the Internet, a shared environment with variable traffic and with great diversity in network bandwidth and host processing speed. We use a novel toolkit approach to build software feedback mechanisms for client/server synchronization, dynamic Quality-of-Service control, and system adaptiveness. Our experimental results show that the feedback mechanisms are effective, and that the player performs very well in the Internet environment.

1 Introduction

Modern workstation and network technology has made software-only solutions for real-time playback of compressed continuous video and audio feasible, even across the Internet. The Internet environment is characterised by the lack of a common clock, wide-spread resource sharing, dynamic workload, and great diversity in host processing speed and network bandwidth. To meet the strict timing requirements of distributed multimedia presentation in the face of such characteristics requires new approaches to client/server synchronization, Quality-of-Service (QoS) control and system adaptiveness.

One approach to client/server synchronization is to use an external mechanism, such as the Network Time Protocol (NTP) [4], to build the illusion of a common clock. However, since such protocols are not ubiquitous, and are rarely engaged between geographically separate sites, it is currently necessary for distributed real-time applications to implement their own mechanisms for maintaining client/server synchronization. Similarly, it has been shown that QoS guarantees can be provided with admission control and resource reservation. However, such approaches are not possible in the current Internet environment, and are still far from being supported in commercial operating systems.

* This project is supported in part by grants from ARPA and the National Science Foundation, and donations from Tektronix, Hewlett-Packard and the Portland Trail Blazers.

This paper explores the use of *software feedback* [5, 6] for client/server synchronization, dynamic QoS control and system adaptiveness in the Internet environment. Software feedback mechanisms already exist in many forms, such as the flow control mechanism used in TCP [2], the clock synchronization mechanism used in NTP [4], and Rowe's video stream frame rate control mechanism [8]. However, most existing approaches are implemented in an ad-hoc manner, and are usually hard-coded for a particular application. Consequently, they suffer from arbitrary structure, hard-to-predict behavior, and wasted effort due to repeated design and implementation of logically similar components. We are developing a toolkit-based approach [6] in order to overcome these drawbacks.

To study the effectiveness of software feedback mechanisms for client/server synchronization, dynamic QoS control and system adaptiveness, and to investigate the toolkit approach, we have constructed a distributed real-time MPEG video and audio player. The player consists of a client and audio and video servers which can be distributed across the Internet. It supports variable play speed and random positioning as well as common VCR functions. The salient features of the player include: (a) real-time, synchronized playback of MPEG video and audio streams, (b) user specification of desired presentation quality, (c) QoS adaptation to variations in the environment, and (d) a toolkit approach to building software feedback mechanisms.

This paper presents the design, implementation and experimental analysis of the player and the software feedback mechanisms. Section 2 describes the overall system architecture. Section 3 discusses the software feedback mechanisms used in the player. Section 4 outlines the implementation. Section 5 presents performance results. Section 6 outlines related work. Finally, Section 7 discusses future work and concludes the paper.

2 System Architecture

Figure 1 shows the architecture of the player. The player has five components: a video server (VS), an audio server (AS), a client, and video and audio output devices. VS manages video streams. AS manages audio streams. The client is composed of a video decoder and a controller which controls playback of both video audio streams and provides a user-interface. The client, VS and AS reside on different hosts, communicating via network. Video and audio output devices reside on the same host or high speed local area network as the client.

A program for the player is a video and audio stream pair: <video-host:video-file-path, audio-host:audio-file-path>, where a video stream is a sequence of frames, and an audio stream is a sequence of samples. These two streams are recorded strictly synchronously. We refer to a contiguous subsequence of audio samples corresponding a video frame as an *audio block*. Therefore, there is a one-to-one correspondence between video frames and audio blocks.

During playback of a program, VS and AS retrieve the video and audio streams from their storage and send them to the client at a specified speed. The

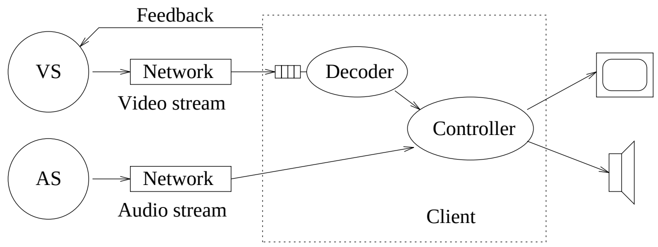


Fig. 1. Architecture of the player

client buffers the streams to remove network jitter, decodes video frames, resamples audio, and plays them to the video and audio output devices respectively.

Programs can be played back at variable speed. *Play speed* is specified in terms of frames-per-second (fps). The player plays a program in real-time by mapping its logical time (defined by sequence numbers for each frame/block) into system time (real time, in seconds) on the client's host machine. Suppose the system time at which frame(i) is displayed is T_i, and the current play speed is P fps, then the time at which frame($i+1$) is played is $T_{i+1} = T_i + \frac{1}{P}$. VS and AS also map the program's logical time into their own system time during the retrieval of the media streams. Synchronization between audio and video streams is maintained at the client by playing audio blocks and displaying video frames with the same sequence number at the same time.

If any stage of the video pipeline, from VS through the network and client buffer to the decoder, does not have sufficient resource to support the current QoS specification it can decide independently to drop frames. The controller of the client also drops late frames (frames which arrive after their display time). A similar approach is implemented for the audio pipeline.

One metric to measure the actual QoS level of the player is its video *display frame rate*. The display frame rate is the number of frames-per-second displayed by the client. Display frame rate should not be confused with play speed, which is also specified in frames-per-second. A valid display frame rate is always equal to or lower than the current play speed. For example, Suppose in a playback, the play speed is P fps, and the display frame rate is F fps ($0 \leq F \leq P$), then $\frac{P-F}{P} * 100\%$ of all frames are dropped by the player.

The maximum actual QoS level supported by a pipeline is referred to as its *effective bandwidth*. In a dynamic system, the effective bandwidth changes with the current system load level. Since our player adapts to changes in effective bandwidth, it can be classified as taking a *best effort* approach to maintaining QoS. However, the player's interface allows users to specify desired QoS levels for video and audio playback. This approach allows users to trade presentation quality for reduced resource consumption, which is useful in a shared environment with limited and diverse resources.

User QoS specification is currently restricted to a single dimension, a desired display frame rate. Future versions of the player will support QoS specifications in several other dimensions [11]. The player tries to yield a display frame rate up to the user-specified frame rate. To comply with the user-specified frame rate, the player drops excess frames at the source of the pipeline, i.e., VS does not retrieve them from storage. VS also spaces the dropped frames evenly throughout the video stream.

We assume that the video and audio devices are very close to the client and hence delay from the client to the output devices can be ignored. We also assume that the mapping from logical time to client system time is precise. In practice, these assumptions are reasonable, especially if the client controller runs with real-time priority. However, a number of other serious problems still remain to be solved. These problems include client/server clock drift, insufficient effective bandwidth to meet the user-specified QoS, and stalls and skips in the pipeline. The next section introduces the basic concepts behind software feedback and shows how it can be used to solve these problems.

3 Software Feedback for Synchronization & QoS Control

Software feedback is a technique that uses feedback mechanisms similar to hardware feedback such as phase-lock loops in control systems [1, 5]. A feedback mechanism monitors the output or internal state of the system under control, compares it to the goal specification, and feeds the difference back to adjust the behavior of the system itself. One beneficial property of feedback mechanisms is that they can control complex systems even when they have only partial knowledge of the system's internal structure. This property suggests that they could be useful in controlling systems, such as our player, that must operate in highly complex and unpredictable environment, such as the Internet.

3.1 A Toolkit Approach to Software Feedback

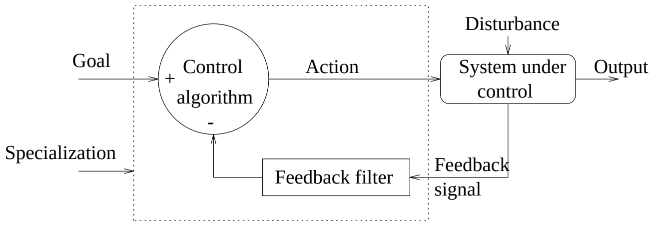


Fig. 2. Software feedback system structure

The structure of a software feedback mechanism is shown in Fig. 2. The mechanism has two basic components: a feedback filter and a control algorithm. The output or internal state of the system under control is measured to get a feedback signal. This signal is then input to the filter to eliminate transient noise.

The control algorithm compares the output of the feedback filter against a goal specification, and takes action to adjust the system under control to keep its behavior within the specification.

A software feedback toolkit includes a set of basic filters and control algorithms, building blocks that are well specified and understood. Filters and control algorithms for a specific feedback mechanism can then be composed from these building blocks. The filters and control algorithms can also be specialized by changing parameter values or modifying structures. In a large feedback system, this specialization might be the result of actions taken by other feedback mechanisms. Hence, the toolkit enables the construction of a potentially large network of interacting feedback mechanisms.

The set of basic filters can include low-pass filters, differential filters, integration filters etc. Consider an input sequence $input[i]$ $(i \geq 0)$. The output sequence $lowpass[i] (i \geq 0)$ of the low-pass filter described by Massalin [5], with a parameter R $(0 \leq R \leq 1.0)$, is defined as:

$$\begin{aligned} lowpass[0] &= input[0] \\ lowpass[i+1] &= (1.0 - R) * lowpass[i] + R * input[i] \qquad \text{where } i \geq 0 \end{aligned}$$

Here the output of the low-pass filter is actually the weighted sum of recent inputs with an aging factor. The construction of filters and control algorithms is the subject of ongoing research, and will appear in future work.

3.2 Software Feedback for Client/Server Synchronization

VS works ahead of the client to mask the video pipeline delay, and a buffer at the client side removes network delay jitter. The remaining client/server synchronization problems are: (a) the server and client system clocks may not be running at exactly the same rate, causing the client and server logical clocks to drift apart and the client buffer to eventually overflow or become empty, (b) the VS logical clock may skip or stall causing a permanent drop or rise in the fill level of the client buffer, and (c) the work-ahead time of VS may be unnecessarily large, reducing player responsiveness and consuming more client buffer space than necessary. We propose a software feedback mechanism which solves these synchronization problems and adapts the player to its dynamic environment.

The synchronization feedback mechanism is implemented in the client, as shown in Fig. 3. It measures the current client time, T_{c}, and the server time, T_{s}, as observed at the client, and computes the raw server work ahead time, $T_{\mathrm{rswa}} = T_{\mathrm{s}} - T_{\mathrm{c}}$. T_{rswa} is input to a low-pass filter, F_1, to eliminate high frequency jitter and get the server work ahead time, T_{swa}. The control algorithm then compares T_{swa} with the target server work ahead time, T_{tswa}, and takes action accordingly.

T_{twsa} in turn is determined by the current network delay jitter level. The jitter of the measured current server work ahead time, $|T_{\mathrm{rswa}} - T_{\mathrm{swa}}|$, is fed to another low-pass filter, F_2, to get the network delay jitter, J_{net}. J_{net} is then used

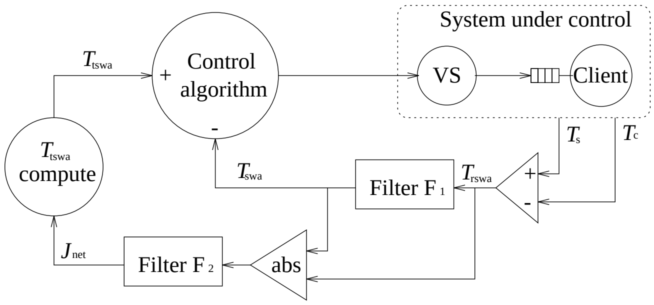


Fig. 3. Structure of the synchronization feedback mechanism

Event		Feedback Action
T_{swa} too low	$T_{swa} < \frac{1}{2}T_{tswa}$	Speed up C_s rate or skip C_s
T_{swa} too high	$T_{swa} > \frac{3}{2}T_{tswa}$	Slow down C_s rate or stall C_s
T_{tswa} too low	$T_{tswa} < \frac{1}{4}K * J_{net}$	Double T_{tswa}
T_{tswa} too high	$T_{tswa} > K * J_{net}$	Halve T_{tswa}

Table 1. Functionality of the synchronization feedback mechanism

to compute T_{tswa}. We get a composite jitter level filter by composing the basic low-pass filters F_1 and F_2.

Table 1 describes the functionality of the synchronization feedback mechanism. C_s refers to the VS clock, and $K > 0$ is a constant. Whenever the control algorithm detects that T_{swa} has deviated too far from T_{tswa}, it adjusts the VS clock rate or skips or stalls it for a certain amount of time, to bring T_{swa} back to T_{tswa}. The decision to adjust the rate or stall/skip the VS clock is based on the rate of deviation. Each time the VS clock is adjusted, the mechanism backs off for a certain amount of time (which is a function of T_{tswa}) to let the effect of the adjustment propagate back to the feedback signal input. T_{tswa} is re-specialized exponentially according to the current network delay jitter level J_{net}. T_{tswa}, filter parameters, back-off time, etc. are also re-specialized upon play speed change.

3.3 Software Feedback for QoS control

The user may specify a desired presentation quality. However, the specified QoS could be greater than the effective bandwidth of the video pipeline. In this case, the pipeline will be overloaded, bottle-neck stages will drop frames randomly, and resources such as server/client processing power and network bandwidth are wasted processing or transmitting the frames that will never be displayed. Further more, the QoS yielded by an overloaded pipeline is usually worse than that yielded by a fully-loaded one. Rather than relying solely on intermediate pipeline stages to control congestion by dropping frames at random, we feed back the QoS observed at the client to VS and allow VS to drop frames intelligently at the source of the pipeline.

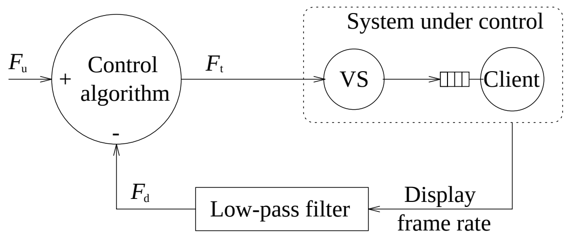


Fig. 4. Structure of the QoS control feedback mechanism

Event		Feedback Action
Pipeline over-loaded	$F_d < F_t - T_h$	$F_t = F_t - \triangle$
Pipeline under-loaded	$F_d > F_t - T_l$ and $F_d < F_u$	$F_t = Min(F_t + \triangle, F_d)$

Table 2. Functionality of the QoS control feedback mechanism

The QoS control feedback mechanism is also implemented in the client, as shown in Fig. 4. Initially, the target frame rate, F_t, at which VS sends frames is set to the user-specified frame rate, F_u. The feedback mechanism monitors the display frame rate at the client and uses a low-pass filter to remove transient noise. The filtered display frame rate, F_d, is then compared against F_u and the existing F_t by the control algorithm. If the pipeline is found to be under- or over-loaded, a new F_t value is computed and fed back to VS.

The control algorithm adjusts F_t linearly. The functionality of the feedback mechanism is described in Table 2. T_l, T_h, and $\triangle$ are three parameters: low and high thresholds and adjust step, where $T_l > 0$, $T_h > 0$, $\triangle > 0$ and $T_h - T_l > \triangle$. These parameters, as well as the back-off time after a feedback action, are re-specialized upon play speed change. The back-off time is also adapted to T_{swa} measured in the synchronization feedback mechanism.

4 Implementation

The player is written in C, using code modified from the Berkeley MPEG decoder [7], and a Motif interface based on a modified version from the University of Minnesota. The streams supported by the player are MPEG-1 video and 8-bit $8K$ sample rate μ-law audio. The software is publicly available [3].

Figure 5 shows the structure of the player. VS and AS run as daemons on their respective hosts. The client is a set of collaborating processes which communicate via shared memory, semaphores, pipes and signals. The video and audio output devices are X Window and AudioFile [10] processes respectively.

The client sends control messages to AS and retrieves the audio stream through a TCP channel. Synchronization between the client and AS is maintained by the TCP flow control mechanism. We assume that the TCP channel has enough bandwidth to support the user-specified audio playback quality.

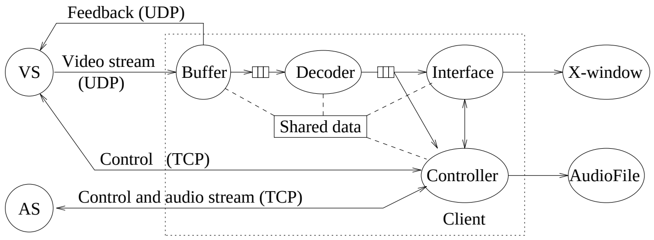


Fig. 5. Process oriented structure of the player

VS sends video frames to the client via UDP, chopping big frames into pieces to comply with UDP packet size limit. The client buffer process reassembles the frames before passing them to the decoder process. To accommodate the Motif programming interface, a separate process from the controller is used to drive the user interface and the display of video frames. VS, AS and the client controller processes run at real-time priority. Logical play time is mapped into system time via the UNIX interval timer.

5 Experiment Results

Several experiments were conducted to evaluate the performance of the player and to study the effectiveness of the feedback mechanisms. We used short clips of action video and audio from basketball games for the experiments. The AS, VS and client processes were run on various architectures (Sun SPARC, HP 9000, and i486). We also tested the player over the Internet with the client at OGI and the servers 28 hops away at Georgia Tech.

In all experiments, our player maintained good audio/video synchronization at the specified play speed. When AS is within our department, even when lots of video frames were dropped due to bandwidth limitations or system overload, audio quality remained high. In the experiments discussed below, the following default configuration was used except when noted otherwise:

- *Servers (VS, AS)*: HP 9000/755 file server.
- *Client*: HP 9000/712.
- *Network*: 10Mb/s Ethernet, server and client on the same subnet.
- *Video stream*: A basketball game clip, frame size 256x192, 9259 frames encoded at 30 fps (about 5 min.), average MPEG frame 2.37KBytes, picture group pattern IBBPBBPBBPBB.

We have defined display frame rate in Sect. 2 as a QoS measurement. However, the display frame rate alone is not sufficient for measuring QoS. Consider two playbacks of the same video stream at the same play speed and display frame rate. If one drops frames more evenly than the other, the former playback will be smoother than the latter. We need a metric to quantify this smoothness aspect of QoS.

One smoothness measure is the deviation of *presentation jitter* [11] from the desired value of zero. Using our assumption that the mapping of logical time (frame number) into system time is precise, and that the delay from the client to the video output can be ignored, we measure the presentation jitter in terms of logical display time.

Consider a video stream of frame sequence $(f_0, f_1, \ldots f_n)$ and a playback displaying a subsequence of these frames: $(f_{i_0}, f_{i_1}, \ldots f_{i_m})$. At each logical display time i ($i \geq 0$ and $i \leq n$), we calculate the logical time error, $e_i = i - i_k$ between the expected frame f_i and the actually displayed frame f_{i_k}, where $i_k \leq i$ and $i_{k+1} > i$, producing the error sequence $E : (e_0, e_1, \ldots e_n)$.

Smoothness S of a playback is the deviation of the sequence E from the perfect playback, which drops no frames and has an error sequence of all zeros. Thus S is defined as:

$$S = \sqrt{\frac{\sum_{e \in E} e^2}{n}}$$

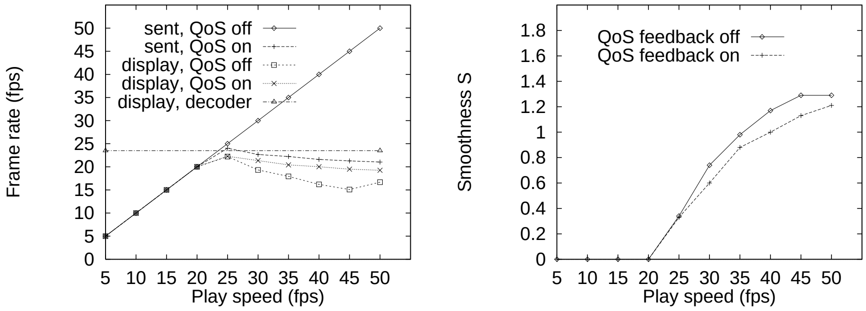
This definition of S is independent of play speed. A lower value of S indicates a smoother playback. S equal to zero denotes perfect playback.

To evaluate the performance of the player and the QoS feedback mechanism, we played the default video stream at various play speeds, at the maximum user-specified frame rate, and synchronization feedback on. Two sets of experiments were done, one with QoS feedback on, and the other with it off. To evaluate the overhead of the player, we also ran the basic Berkeley MPEG decoder, from which our decoder was derived, on the same client host with the same stream. The basic decoder simply plays as fast as possible, without dropping frames. Figure 6(a) shows the frame rate sent by VS and the display frame rate for each of these experiments, and the basic decoder's display frame rate. Comparing the display frame rate curve of our player with QoS feedback on against that of the basic decoder shows that the overhead of our player is 5–20%.

When play speed exceeds 20 fps, the client processor becomes the bottle-neck, and the player consistently yields a higher display frame rate with QoS feedback than without. The rate of frame-drops is simply the difference between sent and displayed frame rates. The frame-drop rate is also consistently lower with QoS feedback, wasting fewer resources. With QoS feedback at any play speed, less then 10% of frames are dropped, while without it, the player drops up to 66% of the frames sent by VS.

Figure 6(b) shows the smoothness measurement S of the two sets of experiments. We see that when the video pipeline is overloaded the player also consistently yields smoother playback with QoS feedback.

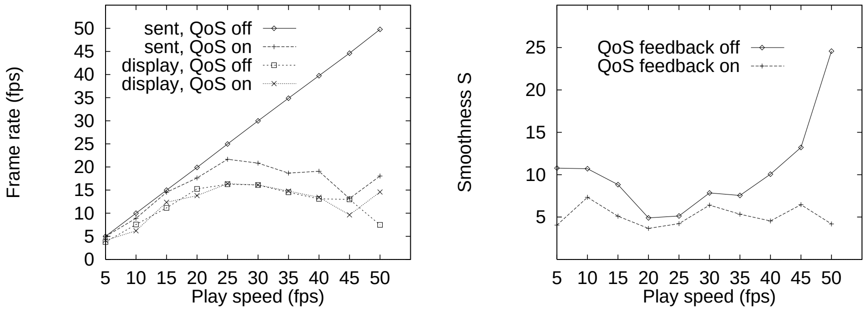
These experiments show that QoS feedback is effective when the client processor becomes the bottle-neck in the video pipeline. This is also true when the network is the bottle-neck. To demonstrate this, we conducted two sets of experiments similar to the ones above, but with VS on a remote Sun SPARC workstation at Georgia Tech. From Fig. 7(a), we see that the Internet bandwidth limits display frame rate to about 15 fps. With QoS feedback, most of the time,



(a) Frame rate sent and displayed (b) Smoothness measurement S

Fig. 6. Comparison with QoS feedback on/off, and with a basic MPEG decoder

we get almost the same display frame rate, but without QoS feedback, up to 83% of the frames sent by VS are dropped, wasting Internet bandwidth. The QoS feedback reduces the frame drop rate to a level of less than 30%. While the QoS feedback does not significantly impact the display frame rate, Fig. 7 shows that it does result in smoother playback.



(a) Frame rate sent and displayed (b) Smoothness measurement S

Fig. 7. Comparison with QoS feedback on/off, with congested network

The system clocks on our server and client workstations are precise and already well synchronized. To demonstrate the effectiveness of the synchronization feedback, we purposely changed the logical to system time mapping on VS to make its logical clock drift from the client's at a rate of about -0.2%. We played the default video stream at a play speed of 30 fps twice, once with synchronization feedback on and once with it off.

Figure 8 shows the server work ahead time, T_{swa}, as measured in the client, against logical play time for the two experiments. In the absence of synchronization feedback, T_{swa} declines gradually from the startup level to zero, at which point the player stalls because all frames arrive too late. Synchronization feedback detects and compensates for the drift, and keeps T_{swa} at a stable level, despite clock drift and clock skip/stall problems. As discussed in Sect.3.2, the synchronization feedback can also adapt target server work ahead time to network delay and delay jitter. In experiments with the default configuration, T_{tswa} was kept at a level where VS worked ahead of the client by about 0.3 seconds. In other experiments with VS on a host at Georgia Tech and the client on our default host, if the Internet was congested VS could work ahead of the client by about 1 second.

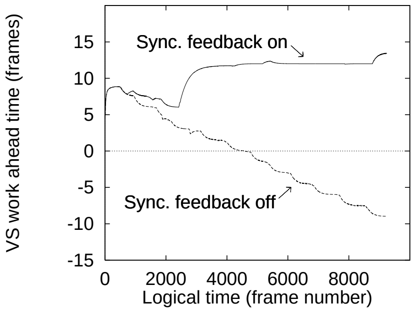


Fig. 8. Synchronization feedback effect

6 Related Work

The Berkeley Continuous Media Player [8, 9] has had the most significant influence on the design and implementation of our player. The Berkeley player and ours use the same MPEG decoder. The Berkeley player uses an ad-hoc software feedback mechanism to adjust the frame rate sent by its server. We follow a toolkit approach, and our feedback mechanism is more efficient and more adaptive to play speed and environmental changes than that used in the Berkeley player as shown by comparing Fig. 6 to similar figures in [9]. However, the Berkeley player relies on NTP [4] for client/server synchronization, while our feedback approach to synchronization is built into the player and does not need external synchronization. Thus our player is more robust and adaptive in Internet environments. Finally, our player supports user-specified QoS.

The idea of software feedback was identified in the Synthesis operating system project [5]. Massalin and Pu demonstrated that software feedback can be used in process scheduling to adapt quickly to system changes. Pu and Fuhrer [6] proposed a toolkit approach to software feedback. Our work furthers the idea of a toolkit approach to software feedback by applying it in our player to solve real-world problems of client/server synchronization, dynamic QoS control and system adaptiveness.

7 Discussion and Future Work

The design, implementation and evaluation of a distributed real-time MPEG video and audio player has been presented. We also discussed a toolkit approach to building software feedback mechanisms for client/server synchronization, dynamic QoS control and system adaptiveness. Our experimental results demonstrate that these feedback mechanisms are effective. With these mechanisms, our player can adapt its QoS to variations in processor speed, network bandwidth and system workload. It can also adapt to variations in network delay and delay jitter, and can compensate for client/server clock drift. These mechanisms make our player robust and allow it to perform well across the Internet.

Our experiences with building feedback mechanisms in the player suggests that the toolkit approach is useful. It leads to a better understanding of system behavior, clearer system structure and design and code reuse. Future research will further refine and evaluate the feedback mechanisms for the player, and further explore the idea of a software feedback toolkit. We will apply this toolkit approach to real-time scheduling, another important consideration for multimedia applications. We also plan to extend our player to incorporate user specification of QoS along more dimensions including spatial and temporal resolution, color quantization, and synchronization accuracy.

References

1. William L. Brogan. Modern Control Theory. Quantum Publishers, Inc. 1974.
2. Lawrence S. Brakmo et. al. TCP vegas: New Techniques for Congestion Detection and Avoidance. Proc. SIGCOMM'94 Symposium, pages 24-35. August 1994.
3. Shanwei Cen. A Distributed Real-Time MPEG Video Audio Player. Available via anonymous FTP from `ftp://ftp.cse.ogi.edu/pub/dsrg/Player`.
4. D. L. Mills. Network Time Protocol (Version 3) Specification, Implementation and Analysis. DARPA Network Working Group Report RFC-1305. University of Delaware, March 1992.
5. H. Massalin and C. Pu. Fine-Grain Adaptive Scheduling Using Feedback. Computing System, 3(1):139-173, Winter 1990.
6. Calton Pu and Robert M. Fuhrer. Feedback-Based Scheduling: a Toolbox Approach. Fourth Workshop on Workstation Operating Systems. Oct. 14-15, 1993.
7. Ketan Patel et. al. Performance of a Software MPEG Video Decoder. ACM multimedia'93, Anaheim, California. August 1993.
8. Lawrence A. Rowe and Brian C. Smith. A Continuous Media Player. Proc. 3rd NOSSDAV. San Diego, California. November 1992.
9. Lawrence A. Rowe et. al. MPEG Video in Software: Representation, Transmission and Playback. Symp. on Elec. Imaging Sci. & Tech., San Jose, CA, February 1994.
10. Thomas M. Levergood et. al. AudioFile: a Network-Transparent System for Distributed Audio Applications. Proc. the USENIX Summer Conference, June, 1993.
11. Richard Staehli, Jonathan Walpole and David Maier. Quality of Service Specifications for Multimedia Presentations. To appear in Multimedia Systems. August, 1995.