Developing PDA for Low-Bitrate Low-Delay Video Delivery

K.-W. Roger Cheuk S.-H. Gary Chan
K.-W. Alice Mong C.-M. Martin Lee S.-S. Shirley Sy
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Kowloon
Hong Kong
Email: {rcheuk, gchan}@cs.ust.hk

Abstract— **Pervasive wireless multimedia applications often require Personal Digital Assistants (PDAs) for processing and playback. The capability of PDAs, however, are generally much lower than desktop PCs. When these devices are used to play back video delivered over a network from a desktop server, their buffers can easily overflow, seriously degrading the video quality. In this paper, we report our implementation of some special stream processing techniques to deal with the capability mismatch between a PC and PDAs for low-delay live video streaming. These techniques are, the Selective Packet Drop (SPD) algorithm, the Game API (GAPI) optimization and the speed adaptation algorithm. All of them can be easily implemented. We show that our system provides much better video quality than systems without our techniques.**

I. Introduction

With their continuous reduction in size and weight, Personal Digital Assistants (PDAs) such as Pocket PCs and Palms have grown in popularity over the past years. Nowadays, PDAs offer multimedia capabilities, network functionalities, and a suite of programming kits for application development.

We have developed a low-delay real-time video delivery system for pervasive multimedia applications such as surveillance, interactive distance learning, video conferencing and so on. It is based on packet-based video, and users roaming in a wireless LAN can ubiquitously access and display video streams with their PDAs at any time. In the system, a desktop PC captures the video, compresses it into the H.263+ format and finally encrypts the compressed video with a key (we have used the RC4 symmetric key encryption algorithm due to its low processing requirement) before streaming through a wireless LAN using UDP to PDAs for display, as illustrated in Fig. 1. Users are able to view the compressed video with a Pocket PC (we have used Pocket PC instead of Palm due to its higher processing and multimedia capabilities).

Streaming video over the Internet and wireless medium presents many challenges [1], [2]. Specifically, PDAs has the following limitations:

- Limited buffering capability — PDAs do not have as large storage as desktops. As a result, how its limited buffer is managed impacts greatly on the system performance. As a matter of fact, in low-delay real-time video, buffering should not be large. When a buffer becomes full, packets have to be dropped or discarded intelligently in order to guarantee high video quality.
- Relatively low processor speed — Despite the continuous improvement in processor speed, PDAs still lag far behind desktops or laptops. This limitation becomes apparent in processing-intensive low-delay video applications.

The relatively low processing speed of PDAs often causes serious degradation in video quality delivered to users, due to the so-called "Speed Mismatch Problem": although video encoding is generally more computationally-intensive than video decoding, the video bandwidth can easily overwhelm the mobile devices, if the encoding rate is not properly chosen or controlled. When this happens, the client's buffer tends to overflow, leading to bursty packet drops. To make the matter worse, many video compression standards such as H.263+ use inter-coded P frames to reduce temporal redundancy in order to conserve bandwidth. As the packets corresponding to these P-frames are dropped, errors propagate to the subsequent frames, leading to poor video quality. We show in Fig. 2 what happens when the encoding frame rate is much higher than the decoding frame rate if the client employs a simple drop-tail buffering policy. The server is a PIII PC with 128MB RAM and the client is an iPaq 3600 Pocket PC. As the server "outpaces" the PDA, high packet drop rate and error propagation result. Clearly, the video quality is unacceptable. This example shows that addressing the mismatch problem is important to offer quality video.

This work was supported, in part, by the Sino Software Research Institute at the HKUST (SSRI01/02.EG21) and the Competitive Earmarked Research Grant from the Hong Kong Research Grant Council (HKUST 6014/01E and HKUST 6199/02E).

Correspondence to: Dr. S.-H. Chan

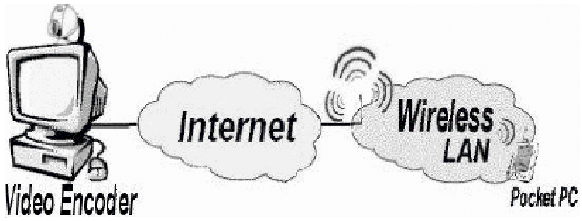

Fig. 1. Our wireless video delivery system.

Fig. 2. Video quality without applying any stream processing techniques.

To address this problem, we have implemented a number of special processing techniques. Our techniques are general enough to work with many error recovery schemes (such as [3], [4], [5] and references therein) and many network-adaptive rate control schemes [6], [7]. Regarding video, we have used a GOP structure of IPPPP...IPPPP..., with the GOP size chosen so as to tradeoff startup delay, error propagation and bandwidth requirement (we have used a GOP size of 10 in this paper). The techniques we have implemented in the system are:

- Client-side implementation:
  - Selective Packet Drop (SPD) algorithm: SPD drops packets to retain only the most important few frames (i.e., the leading P-frames) in the buffer upon the arrival of an I-frame. This is done in order to prevent buffer overflow and to synchronize the video delivered between a server and the clients.
  - Game API optimization: In order to improve the decoding rate of PDA, we have used the Game API library for fast video frame display. We demonstrate that this technique can markedly increase the decoding rate as compared to simply using Win32 system calls. The processing gap between a desktop PC and a Pocket PC is narrowed due to reduced CPU usage in rendering activities.
- Feedback-based implementation: Note that even with SPD and GAPI, the video may appear "jerky" if a server outpaces a receiver by a large margin (mainly due to the fact that whenever an I-frame arrives, a burst of P-frames may be dropped.). In order to address this problem, the server can match the clients' processing capabilities based on feedback from clients. Clearly,

due to the feedback implosion issue, this solution is suitable for unicast and small-scale multicast applications (for large-scale multicast applications, a means of controlling the feedback rate has been suggested in [8]).
Adapting the encoding rate to the decoding rate has been discussed in [9], [10], [11]. Most of these algorithms require a receiver periodically and continuously feeding back its buffer state back to the server. This increases the network bandwidth and may cause the server to adapt too frequently, which is visually disagreeable. We implement a new algorithm in our system in which a client only updates the server whenever it is necessary. The speed matching algorithm is termed *Speed Adaptation Algorithm* (SA).
In SA, clients measure the decoding time it takes to decode its frames. Each client measures the decoding time of the first frame and reports a slightly increased decoding time as the target inter-frame time to the server (via a feedback TCP channel). Hence after, each client compute at each frame a new target inter-frame time based on the measured frame decoding time. Whenever there is a significant change in the target inter-frame time as compared to the last feedback value, the client updates the server with the new value. Based on this feedback, the server adjusts its encoding frame rate so as to match the speed of the client(s). Obviously, such a solution is effective when the processing capability of the PDAs does not vary much during a round-trip, which is usually the case.

By applying these techniques, we have successfully built a low-delay live video surveillance system with a Pocket PC over a wireless LAN. Our results show that excellent video quality can be achieved on a Pocket PC.

This paper is organized as follows. We first describe in detail these techniques in Sect. II. In Sect. III, we present our experimental environment and measurement results. We conclude in Sect. IV.

II. System Implementation

In this section, we describe in more detail the client-side implementations followed by the feedback-based implementation.

A. Client-side solutions

We have implemented the following two techniques in our video delivery system at the client-side, namely the Selective Packet Drop (SPD) algorithm and the Game API (GAPI) optimization.

A.1 Selective Packet Drop (SPD) algorithm

Note that the importance of video frames in a GOP sequence IPPPP... decreases from the first I frame to the last frame in that GOP. This is because each P frame in the GOP uses the previous frame as the reference frame. The Selective Packet Drop (SPD) algorithm aims to drop the trailing P-frames to prevent a buffer from overflowing while maintaining low video delay.

In SPD, packets are allowed to accumulate in the buffer as long as there is buffer space. To keep the delay low and to take into consideration of network delay jitter and fluctuations in the time required to encode/decode a frame, the n most important video frames (i.e., the leading frames at the head of the buffer queue) are retained in the buffer whenever an I-frame arrives. Clearly, higher network jitter and encoding/decoding variability require higher n, which also means a higher delay.

We use a dedicated thread to run the SPD. The module implemented is shown in Fig. 3. It starts by initializing the queue data structure and the enqueue flag. It then enters into a loop to handle all received packets by the SPD algorithm. When the first packet of an I-frame is received, all except the first n frames in the queue are dropped. The last block of the pseudo code enqueues the currently received packet if the enqueue flag is set. The purpose of the enqueue flag is to make sure that once a frame is dropped, it waits until the next I-frame arrives before accepting packets again as the subsequent P frames would be useless. If the queue cannot accommodate all packets of the frame at the tail of the queue, the function call to `RemoveCurrRecvFrame` is used to remove that frame's packets buffered in the queue.

```
SPDThread(m, n)
 1  Q ← Empty Queue of size m
 2  EnqueueFlag ← 1
 3  while 1
 4  do ReadPacket(P)
 5     if P is the first packet of an I-frame
 6       then Retain(Q, n)
 7            EnqueueFlag ← 1
 8     if EnqueueFlag = 1
 9       then if not Full(Q)
10              then Enqueue(Q, P)
11              else RemoveCurrRecvFrame(Q)
12                   EnqueueFlag ← 0
```

Fig. 3. The Selective Packet Drop (SPD) algorithm.

TABLE I
Some Game API display-related functions.

Function	Description
GXOpenDisplay	Initialize the Game API library
GXCloseDisplay	Release the resources used by Game API
GXBeginDraw	Prepares the Game API before display
GXEndDraw	Release the resources allocated with GXBeginDraw after display is done
GXGetDisplayProperties	Obtain the hardware display capabilities usable by the Game API library

A.2 Game API Optimization

Microsoft has released a new set of API, the Game API (GAPI), specifically targeted for use in high-performance, real-time games on Pocket PC devices. It has a number of useful functions for fast video display. We have achieved a substantial increase in the decoding frame rate (nearly 40% in our experiments) when we change our display code from the Win32 graphics API to this set of GAPI. This improvement comes from the fact that these functions allow the display memory to be accessed directly. With GAPI, the decoding rate is significantly increased. Table I shows the list of the display-related functions available in the Game API.

B. Feedback-based solution

B.1 System Description

We show in Fig. 4 the system diagram of our server-side video-streaming subsystem with the feedback and speed adaptation architecture. It consists of the following components: the Synchronization Unit, the H.263+ encoder, the Network Module, the Token Bucket and the virtual Clock. The Synchronization Unit controls the rate at which a frame is captured by waiting for at least one token in the Token Bucket. When the unit obtains a picture and passes it to the H.263+ encoder, a token is removed from the token bucket. Tokens are inserted to the token bucket by a clock ticking at a rate specified by the most recent feedback data from the receiver. The clock is informed of the most updated token insertion interval by the network module. When the token bucket is full, no more tokens can be inserted. The network module buffers the compressed stream from the encoder and sends it to the user(s) via UDP. It may also poll the client(s) for feedback.

Regarding the client-side video-streaming subsystem (Fig. 5),

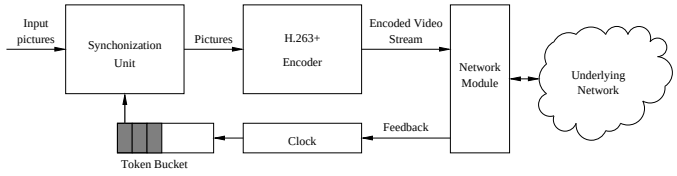

Fig. 4. Server-side video-streaming subsystem diagram.

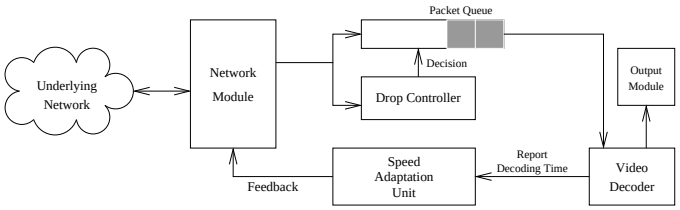

Fig. 5. Client-side video-streaming subsystem diagram.

it has a Packet Queue to buffer all the video packets from the network. The drop controller analyzes all the incoming video packets and decides when to drop packets. It notifies the packet queue of the dropping decision it needs to perform. Independently, the H.263+ video decoder reads packets from the packet queue and decodes them to output to the output module. It also measures the time to decode a frame and sends the data to the Speed Adaptation Unit. Using the data, the Speed Adaptation Unit decides whether to feed back to the server based on a formula given below (in Sect. II-B.2).

B.2 Speed Adaptation

Note that the achievable frame-rate at a client may fluctuate with time (due to background jobs and variations in frame decoding time). In order for the server not to outpace the client most of the time, the interval between successive encoded frames ("inter-frame" time) should not be set simply equal to the average time to decode a frame at the client, but slightly longer to account for the statistical fluctuation of the frame-rate at the client. In our system, each client hence reports to the server a requested inter-frame time using a mechanism similar to the round-trip time estimation as used in TCP, i.e., it first estimates the mean time to decode a frame and the mean deviation of it, and then requests an inter-frame time for the encoding process using these data.

The mean time to decode a frame is estimated as follows. For every frame i, $i \geq 1$, the decoder measures

- $T_t^{(i)}$ — the total time interval between the rendering of frame $i-1$ and frame i;
- $T_r^{(i)}$ — Obviously, $T_t^{(i)}$ consists of two parts: the busy time for the CPU to process the packets of frame i, and the idle time for the CPU to wait for the the constituent packets of frame i to arrive (mainly due to the blocking call used in UDP, `recvfrom`). Denote the total idle time spent in the function call `recvfrom` by $T_r^{(i)}$.

Obviously, the busy time a CPU takes to process the frame i is the limit the PDA can process frame i (the maximum decoding capability at that time), which is given by, $T^{(i)} = T_t^{(i)} - T_r^{(i)}$. $T^{(i)}$ is hence the inter-frame time for frame i which the Pocket PC is capable of processing[1]. The average time $\overline{T^{(i)}}$ to decode and display a frame can therefore be estimated as $\overline{T^{(i)}} = \alpha\overline{T^{(i-1)}} + (1-\alpha)T^{(i)}$, where $0 < \alpha < 1$. Clearly, a high α means that the adaptation "remembers" history and is resistant to changes or fluctuation; on the other hand, a low α means that the system is more susceptible to statistical fluctuation. We estimate the mean deviation of $\overline{T^{(i)}}$, $D_T^{(i)}$, as $D_T^{(i)} = \gamma D_T^{(i-1)} + (1-\gamma)|T^{(i)} - \overline{T^{(i)}}|$, where $0 < \gamma < 1$.

Each value $\overline{T^{(i)}} + \beta D_T^{(i)}$ is a target inter-frame time for feedback, where β is a positive constant. Choosing too large a β may unnecessarily reduce the frame rate and too low a β will increase the chance that the decoder drops packets due to statistical fluctuation (from our experiments, $\beta = 0.8$ performs quite well).

To reduce the amount of feedback, the target inter-frame time is sent selectively, i.e., when the target inter-frame time is significantly different from the previously feed back value. In our implementation, it is feed back when the difference between the current value of $\overline{T^{(i)}} + \beta D_T^{(i)}$ and the previously transmitted feedback value is larger by more than a certain fraction, f. As compared to the Continuous Media Player in which the encoder speed is adjusted by a mechanism similar to Additive Increase and Multiplicative Decrease (AIMD), our speed adaptation algorithm achieves a smoother rate fluctuation and lower feedback overhead [9].

In the server, it adapts the decoding speed by using a token bucket. The time between successive token insertion is equal to the most recent feedback value. The encoder has to obtain a token from the token bucket before encoding a frame; otherwise it is stalled for a token. In this way, the encoding rate is limited to the rate as suggested by the recent feedback data, but is unrestricted if the server's encoding frame rate is less than the achievable decoding frame rate at clients.

[1] We assume that the decoding thread is of highest priority possible, in which the execution time of other preempting threads/processes can be ignored. Other threads/processes are allowed to run when the decoding thread is waiting for a packet.

III. Experimental Results and Comparisons

We have implemented the aforementioned techniques to evaluate the effectiveness of our solutions. In this section, we first present our experimental environment followed by some illustrative measurements and results.

A. Experimental Environment

We have used the Foreman as a representative video sequence in our experiment (the use of other sequences show a similar trend and hence is not discussed). The sequence consists of 400 frames and is in the QCIF format. The frames are encoded in the H.263+ format before delivered over the network. The server-side video delivery program runs on a Pentium III 550MHz PC with 128 MB memory. The server is connected to a 100Mbps LAN. The mobile access point for offering wireless network connections is directly connected to the same LAN. The client-side program runs on an iPaq 3600 Pocket PC. Besides the wireless LAN card, no other additional hardware is installed to the Pocket PC.

Regarding the H.263+ settings, we have used a Quantization Parameter (QP) of 13, a search window size of 31 and a GOP size of 10. Error concealment is not used so as to eliminate its effect for fair comparisons. We use the following values in our experiment: $\alpha = 0.875$, $\beta = 0.8$, $\gamma = 0.75$, $T_a^{(0)} = 100$ms, $D_T^{(0)} = 3$ms, $n = 0$ (low-delay video), and $f = 10\%$.

The encoded video stream is transmitted packet-by-packet to the clients. Each encoded frame is divided into blocks of $1,024$ bytes of UDP packets. The buffer at the decoder side is a FIFO queue accommodating up to 32 packets.

B. Illustrative Measurement Results

The evaluation is done by comparing the picture quality in terms of PSNR with and without our solutions. Note that since some packets may be dropped, we compute PSNR only for the displayed complete frames. We also compare the encoding frame rate with and without Speed Adaptation.

Note that there is one more issue that needs to be considered. As packet loss can be due to reasons other than speed mismatch between the server and a client (i.e., channel errors and congestion losses), it is necessary to isolate their effect from our experiments in order to draw meaningful conclusions. For all experiments, we independently maintain a log for packets transmitted and a log for packets received by the client application. By comparing the logs at the client-side and the server-side for each experiment, it is confirmed that the wireless environment has negligible error and there is no congestion losses throughout any of the experiments. We conclude that packet loss is due to the inability to process packets fast enough before buffer overflow.

B.1 Client-based solutions

In Fig. 6 we show the decoded video quality when none of our solutions is used (i.e., the NO OPTION setting). The gaps in the sequence means that there are dropped frames due to buffer overflow and propagation errors. The video quality decreases quite sharply once packets start to be dropped (at around frame 50). This is when the decoder buffer becomes full (as the encoder out-runs the decoder). With the buffer often being full, we also observe a rather high delay (in seconds) in the system.

The "landsliding" performance in video quality can be markedly improved when our SPD algorithm is used. This is

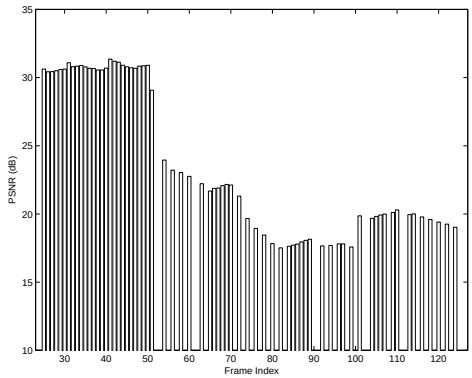

Fig. 6. PSNR for the decoded frames with the NO OPTION setting.

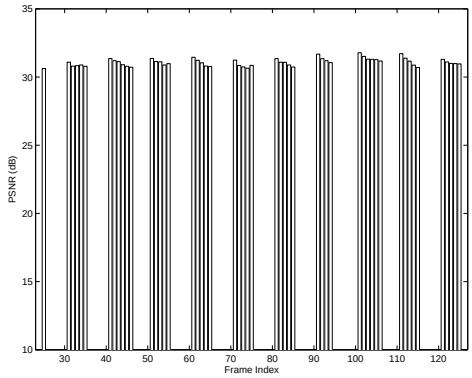

Fig. 7. PSNR of decoded frames with SPD.

shown in Fig. 7. Note that the PSNR is maintained at a high level. Since the arrival of an I-frame triggers the intelligent packet drop algorithm, some video frames are not displayed, as evident from the discontinuities in the figure. Basically, in each GOP, around half of the frames are dropped. Though PSNR and delay are greatly improved, the burst of frame loss leads to some "jerkiness" in the decoded sequence.

This missing frame problem can be solved by applying the GAPI optimization, as shown in Table II. In this table, the results for both cases are obtained (both cases also with the SPD algorithm turned on). By applying GAPI optimizations, the decoding frame rate increases substantially and hence reduce significantly the fraction of frame loss. Note that the combined method, i.e. the SPD algorithm and GAPI optimizations, requires implementation at the client-side only, making it suitable for multicasting video to a large number of PDAs.

B.2 Feedback-based solution

We finally present the performance of a feedback-based system with SPD and GAPI implemented at the client side PDA and the Speed Adaptation algorithm implemented at the server side. We show in Fig. 8 the corresponding PSNR as measured in the PDA. Clearly, the video quality has been greatly improved with much less dropped frames. The figure shows that our speed adaptation algorithm is effective in matching the speeds between the encoder and decoder. The end-to-end delay is also low, due to the synchronizing nature of our Selective Packet Drop algorithm.

TABLE II
Overall frame rate statistics for the 400 frames Foreman sequence

Options	Encoding frame rate (frame/s)	Decoding frame rate (frame/s)	Fraction of frame loss
SPD alone	34.59	17.41	48.25%
SPD with GAPI optimization	34.52	29.32	14.5%

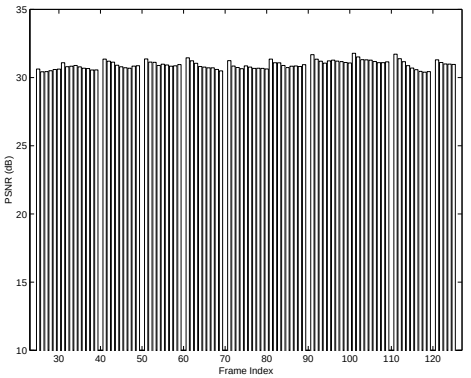

Fig. 8. PSNR of the decoded frames in a feedback-based system, with speed adaptation, SPD and GAPI optimization.

IV. Conclusions

In this paper, we have described our experience in implementing a low-delay video streaming system. Pocket PCs are generally much slower than desktop PCs, and hence the encoder may easily overwhelm the decoder, leading to packet drop and poor video quality (the so-called "Speed Mismatch Problem"). Stream processing techniques are needed to bridge the processing gap between the encoder and the decoder.

This speed mismatch problem can be addressed by our Selective Packet Drop (SPD) algorithm, the Game API optimization and the speed adaptation (SA) algorithm. The SPD algorithm drops the least important frames whenever an I-frame arrives to prevent buffer overflow while keeping the delay low. The Game API optimization renders video with a higher speed, hence narrowing the processing gap. The speed adaptation (SA) algorithm uses clients' feedback to match the encoding rate to the decoding capability of the clients.

Our experiments show that video quality is substantially improved with our techniques. The display frame rate with the Game API can be increased significantly. This shows that our techniques are useful and implementable for pervasive multimedia applications such as surveillance and distance-learning.

References

[1] D. Wu, Y. T. Hou, W. Zhu, Y.-Q. Zhang, and J. M. Peha, "Streaming video over the Internet: approaches and directions," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 11, pp. 282–300, March 2001.

[2] D. Wu, Y. T. Hou, and Y.-Q. Zhang, "Scalable video coding and transport over broadband wireless networks," *Proceedings of the IEEE*, vol. 89, pp. 6–20, January 2001.

[3] T.-W. Lee, S.-H. Chan, Q. Zhang, W.-W. Zhu, and Y.-Q. Zhang, "Allocation of layer bandwidth and FEC for video multicast over wired and wireless networks," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 12, pp. 1059–1070, Dec 2002.

[4] A. Majumda, D. Sachs, I. Kozintsev, K. Ramchandran, and M. Yeung, "Multicast and unicast real-time video streaming over wireless LANs," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 12, pp. 524 –534, June 2002.

[5] B. Girod and N. Farber, "Feedback-based error control for mobile video transmission," *Proceedings of the IEEE*, vol. 87, pp. 1707–1723, October 1999.

[6] Q. Zhang, W. Zhu, and Y.-Q. Zhang, "Network-adaptive rate control with TCP-friendly protocol for multiple video objects," in *Proceedings of IEEE International Conference on Multimedia and Expo*, vol. 2, pp. 1055–1058, July/August 2000.

[7] J. Cabrera, A. Ortega, and J.I.Ronda, "Stochastic rate-control of video coders for wireless channels," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 12, pp. 496 –510, June 2002.

[8] J. C. Bolot, T. Turletti, and I. Wakeman, "Scalable feedback control for multicast viodeo distribution in the Internet," in *Proceedings of ACM SIGCOMM 94*, vol. 24, pp. 58–67, October 1994.

[9] L. A. Rowe and B. C. Smith, "A continuous media player," in *Proceedings of Network and Operating System Support for Digital Audio and Video*, pp. 376–386, 1992.

[10] A. Goel, M. H. Shor, J. Walpole, D. C. Steere, and C. Pu, "Using feedback control for a network and CPU resource management application," in *Proceedings of the 2001 American Control Conference (ACC)*, vol. 4, pp. 2974–2980, June 2001.

[11] S. Cen, C. Pu, R. Staehli, C. Cowan, and J. Walpole, "A distributed real-time MPEG video audio player," in *Proceedings of the 5th International Workshop on Network and Operating System Support for Digital Audio and Video*, pp. 151–162, 1995.