NetMedia: Streaming Multimedia Presentations in Distributed Environments

Aidong Zhang, Yuqing Song, and Markus Mielke
State University of New York at Buffalo

We present the design and implementation of NetMedia, a middleware that supports client–server distributed multimedia applications. In NetMedia, individual clients may access multiple servers to retrieve customized multimedia presentations. Each server simultaneously supports multiple clients. NetMedia has transmission support strategies and robust software systems at both server and client ends.

Researchers have become increasingly interested in flexibly constructing and manipulating heterogeneous presentations from multimedia data resources[1] to support sophisticated applications.[2] In these applications, information from multimedia data resources at one location must be made available at other remote locations for collaborative engineering, educational learning and tutoring, interactive computer-based training, electronic technical manuals, and distributed publishing. Such applications require that we store basic multimedia objects in multimedia databases or files. The multimedia objects—such as audio samples, video clips, images, and animation—are then selectively retrieved, transmitted, and composed for customized presentations. For example, in asynchronous distance learning, we may assemble basic parts of the multimedia presentations stored in the multimedia databases into different presentations for individual learning and training; in medical diagnosis for training and electronic technical manuals, we may assemble a sophisticated presentation containing images, video clips, and audio samples from the data stored in the multimedia databases. Such customized presentations require highly flexible compositions among various multimedia streams.

To support such advanced applications, we need novel approaches to network-based computing to position adaptive quality-of-service (QoS) management as its central architectural principle. One of the key problems is to design the end system software to couple to the network interface. Also, in the best-effort network environment such as the Internet, the transmission of media data may experience large variations in available bandwidth, latency, and latency variance. Consequently, network delays may occur in delivering media data. To meet the demands of the advanced applications over the Internet in the real world, end system software must have the ability to efficiently store, manage, and retrieve multimedia data. We must develop fundamental principles and advanced techniques for designing end-system software integrated with the network to support sophisticated and customized multimedia applications.

Our contribution

This article focuses on the design strategies of NetMedia, middleware for client–server distributed multimedia applications (please see the "Related Work" sidebar for research conducted and developed by others). NetMedia provides services to support synchronized presentations of multimedia data to higher level applications. In the NetMedia environment, an individual client may access multiple servers to retrieve customized multimedia presentations, and each server simultaneously supports multiple clients. NetMedia is capable of flexibly supporting synchronized streaming of continuous media data across best-effort networks. To achieve this, we design a multilevel buffering scheme to control the collaborations between the client and server for flexible data delivery in distributed environments, an end-to-end network delay adaptation protocol (sending and display media units in a stream—termed DSD) to adjust the sending rate for each stream according to the optimum network delay,[3] and the probe-loss utilization streaming (PLUS) protocol, a flow and congestion control protocol that uses network status probing to avoid congestion rather than react to it. The DSD and PLUS schemes integrate transmission support strategies and robust software systems at the server and client ends to dynamically handle adaptive QoS

1070-986X/02/$17.00 © 2002 IEEE

Related Work

Various research has been conducted to develop techniques for distributed multimedia systems. Collaboration between clients and servers has been addressed to support a globally integrated multimedia system.[1] In particular, researchers have studied buffering and feedback control for multimedia data transmission in the distributed environment. In Ramanthan and Rangan's work,[2] they proposed server-centric feedback strategies to maintain synchronized transmission and presentation of media streams. Nahrstedt and Smith[3] used feedback to design robust resource management over the network. Hui et al.[1] designed a backward feedback control approach to maintain loosely-coupled synchronization between a server and a client. Feng et al.[4] developed an approach to use the buffer, a priori information, and the current network bandwidth availability to decide whether one should increase the quality of the video stream when more network bandwidth becomes available. Mielke and Zhang[5] discuss an approach to the integration of buffering and feedback control.

Substantial research has been conducted to reduce the burstiness of variable-bit-rate streams by prefetching data.[6] Researchers have proposed many bandwidth smoothing algorithms specifically for this task. These algorithms smooth the bandwidth requirements for video stream retrieval from the video server and transmission across the network. Given a fixed client buffer, it's possible to minimize the peak bandwidth requirement for the continuous video stream delivery while minimizing the number of bandwidth rate increases.[7] Another scheme aims at minimizing the total number of bandwidth changes.[8] It's also possible to minimize bandwidth variability[9] and buffer utilization[6] or adhere to a buffer residency constraint.[6]

Researchers have also investigated flow and congestion control for multimedia data transmission in best effort networks.[10] These approaches assume no direct support for resource reservation in the network and attempt to adapt the media streams to current network conditions. Best-effort transmission schemes adaptively scale (reduce or increase) the bandwidth requirements of audio and video streams to approximate a connection that's currently sustainable in the network. Best-effort schemes split into TCP-friendly and non-TCP-friendly mechanisms. TCP-friendly algorithms avoid starvation of TCP connections by reducing their bandwidth requirements in case of packet loss as an indicator of congestion.[11] Non-TCP-friendly schemes reduce rates only to guarantee a higher QoS to their own streams, thereby forcing TCP connections to back off when competing for insufficient network bandwidth.[12]

Enforcing synchronization in playout scheduling has also been recognized.[13] Traditionally, servers focus more on keeping resource costs low at the client side. We observe that the enforcement of novel dynamic synchronization algorithms at the client side has become feasible.[13] Also, others have proposed control schemes for processing user interactions[14] for either centralized or distributed environments. The difficulty of interactive support for video streams is the forward motion prediction used by most advanced compression techniques.[15] Several papers propose methods for implementing fast forward but don't deal adequately with fast backward playback. Normally streams are sent out in encoding and not in display order to make it easier for forward playback decoding. This complicates the video frames' decoding in the backward direction and the synchronization between participating streams after an interactive request.

Recent research also addresses the need for a middleware framework in transmitting multimedia data. The middleware is located between the system level (operating system and network protocol) and the applications.[16] Neufeld et al.[17] describe a good conceptual model of the middleware. They propose a real-time middleware for asynchronous transfer mode (ATM) systems. It provides virtual connection setup, bandwidth reservation, and session synchronization. Li and Nahrstedt[18] describe a middleware control structure, which requires applications to implement observation methods for respective system resources and proper adaptation policies within the application.

However, there's a lack of a middleware structure that can effectively put all component techniques together as multimedia middleware services and provide a mapping from the high-level requests of the application to the low-level implementation of the system to support customized multimedia applications. Ideally, the application should be able to treat a multimedia stream like a file and access it without worrying about synchronization, buffer management, and transmission over the network. In contrast to the traditional networked file server, multimedia computing introduces two new problems. First, the size of multimedia objects requires large aggregate I/O bandwidth. The bandwidth also must be large enough to deal with the burstiness introduced by compression techniques. The multimedia middleware must be able to handle bursty and voluminous data at high speeds. Second, the service must be guaranteed timely to ensure smooth playback at the client display. Main memory must be efficiently managed for staging data between the disks and the network. To accommodate different variations in bit rates for compressed streams as well as some-

continued on p. 58

management for customized multimedia presentations. We use these schemes to enforce interactive control (such as fast forward or backward playback, pause, seek, and slow motion) with immediate response for all multimedia streams. To the best of our knowledge, this is the first work that provides an integrated solution to buffer management, congestion control, end-to-end delay variations, interactivity support, and stream synchronization.

continued from p. 57

what unpredictable transmission and retrieval delays, we also need novel end-to-end control approaches. Moreover, the system must be able to respond flexibly and dynamically to various types of user interactions.

References

1. J. Hui et al., "Client–Server Synchronization and Buffering for Variable Rate Multimedia Retrievals," *IEEE J. Selected Areas in Comm.*, vol. 14, no. 1, Jan. 1996, pp. 226-237.
2. S. Ramanathan and P.V. Rangan, "Feedback Techniques for Intra-Media Continuity and Inter-Media Synchronization in Distributed Multimedia Systems," *The Computer J.*, vol. 36, no. 1, 1993, pp. 19-31.
3. K. Nahrstedt and J. Smith, "New Algorithms for Admission Control and Scheduling to Support Multimedia Feedback Remote Control Applications," *Proc. 3rd IEEE Int'l Conf. Multimedia Computing and Systems* (ICMCS 96), IEEE CS Press, Los Alamitos, Calif., 1996, pp. 532-539.
4. W. Feng, B. Krishnaswami, and A. Prabhudev, "Proactive Buffer Management for the Streamed Delivery of Stored Video," *Proc. ACM Multimedia*, ACM Press, New York, 1998, pp. 285-290.
5. M. Mielke and A. Zhang, "A Multi-Level Buffering and Feedback Scheme for Distributed Multimedia Presentation Systems," *Proc. 7th Int'l Conf. Computer Comm. and Networks* (IC3N 98), IEEE CS Press, Los Alamitos, Calif., 1998, pp. 219-226.
6. W. Feng, *Buffering Techniques for Delivery of Compressed Video in Video-on-Demand Systems*, Kluwer Academic, New York, 1997.
7. W. Feng and S. Sechrest, "Smoothing and Buffering for Delivery of Prerecorded Compressed Video," *Proc. SPIE Multimedia Computing and Networking*, SPIE Press, Bellingham, Wash., 1995, pp. 234-242.
8. W. Feng, F. Jahanian, and S. Sechrest, "An Optimal Bandwidth Allocation Strategy for the Delivery of Compressed Prerecorded Video," *ACM/Springer-Verlag Multimedia Systems J.*, vol. 5, no. 5, 1997, pp. 297-309.
9. J. Salehi et al., "Optimal Buffering for the Delivery of Compressed Prerecorded Video," *Proc. ACM Special Interest Group on Computer/Communication System Performance* (Sigmetrics 96), ACM Press, New York, 1996, pp. 222-231.
10. I. Busse, B. Deffner, and H. Schulzrinne, "Dynamic QoS Control of Multimedia Applications Based on RTP," *Computer Comm.*, vol. 19, no. 1, Jan. 1996, pp. 49-58.
11. D.D. Clark, S. Shenker, and L. Zhang, "Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanism," *Proc. ACM Special Interest Group on Data Comm.* (Sigcom 92), ACM Press, New York, 1992, pp. 14-26.
12. S. Chakrabarti and R. Wang, "Adaptive Control for Packet Video," *IEEE Multimedia and Computer Systems*, IEEE CS Press, Los Alamitos, Calif., 1994, pp. 56-62.
13. T.V. Johnson and A. Zhang, "Dynamic Playout Scheduling Algorithms for Continuous Multimedia Streams," *ACM Multimedia Systems*, vol. 7, no. 4, July 1999, pp. 312-325.
14. N. Hirzalla, B. Falchuk, and A. Karmouch, "A Temporal Model for Interactive Multimedia Scenario," *IEEE MultiMedia*, vol. 2, no. 3, Fall 1995, pp. 24-31.
15. M.S. Chen and D. Kandalur, "Downloading and Stream Conversion: Supporting Interactive Playout of Videos in a Client Station," *Proc. IEEE Multimedia Computing and Systems*, IEEE CS Press, Los Alamitos, Calif., 1995, pp. 73-80.
16. P.A. Bernstein, "Middleware: A Model for Distributed System Services," *Comm. ACM*, vol. 39, no. 2, Feb. 1996, pp. 86-98.
17. G.Neufeld, D.Makaroff, and N.Hutchinson, *The Design of a Variable Bit Rate Continuous Media Server*, tech. report TR-95-06, Dept. Computer Science, Univ. of British Columbia, Vancouver, Canada, March 1995.
18. B. Li and K. Nahrstedt, "A Control-Based Middleware Framework for Quality of Service Adaptations," *IEEE J. Selected Areas in Comm.*, vol. 17, no. 9, Sept. 1999, pp. 1632-1650.

NetMedia architecture

We illustrate the overall system architecture in Figure 1. The system has three main components: the client for presentation scheduling, the server for resource scheduling, and the database (file) systems for data management and storage. Each client supports a graphical user interface (GUI) and synchronizes images and audio and video packets and delivers them to an output device such as a PC or a workstation. Meanwhile, each server is superimposed on a database system and supports the multiuser aspect of media data caching and scheduling. It maintains timely media data retrieval from the media database and transfers the data to the client sites through a network. Finally, each database system manages the insertion, deletion, and update of the media data stored in the local database. Because certain media streams may be represented and stored in different formats, the underlying database systems can be heterogeneous. The aim in the proposed middleware is to give users as much flexibility and individual control as possible. The middleware must support each media type individually to allow independent and interactive functionalities as well as QoS requirements.

Server design

The server design permits individual stream access with support of sharing resources to enhance scalability. We can summarize the aim of server functionality in the following points:

- resource sharing (use of common server disk reading for multiple clients),
- scalability (support multiple front end modules),
- individual QoS and congestion control (managing individual streams), and
- interactivity.

We divide the server and client design into the front end and the back end. The server back end includes the disk service module that fills in the server buffer according to the request. The server buffer contains a sub-buffer for each stream. All streams and front-end modules share the back-end module.

The server front end includes communication and packetization modules that support reading data from the server buffer and delivering it to a network according to the QoS requirements for each stream. It also deals with the clients' admission control.

Figure 2 shows the server component's implementation design for processing audio and video streams. The disk service module is realized through DiskReadThread. The packetization module is realized by SendThread, which reads media units from the server buffer and sends packets to the network. The communication module is realized through ServerTimeThread, AdmissionThread, and AdmittedClientSet. ServerTimeThread reports the server's current time and estimates the network delay and time difference between server and client. AdmittedClientSet keeps track of all admitted clients. The system uses Server_Probe_Thread to get feedback messages and control messages from the probe thread at the client site and initiates control of probing the network.

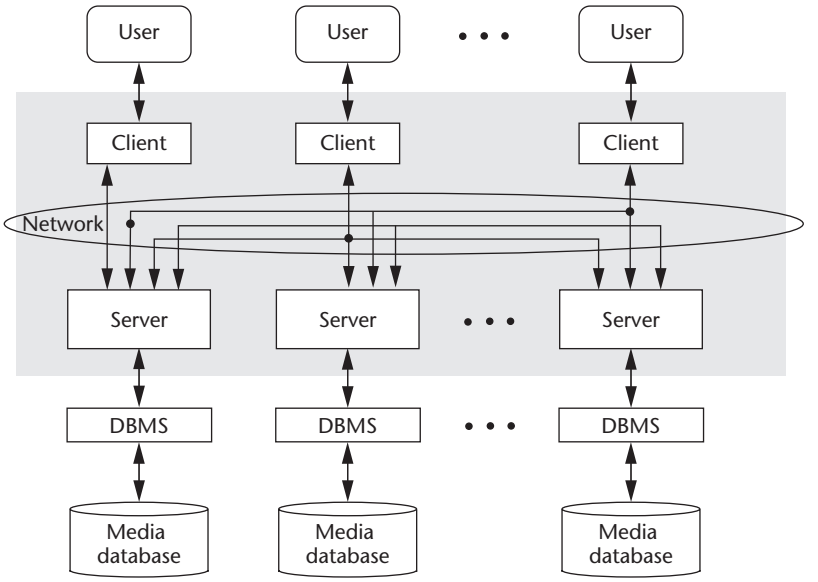

Figure 1. NetMedia system architecture.

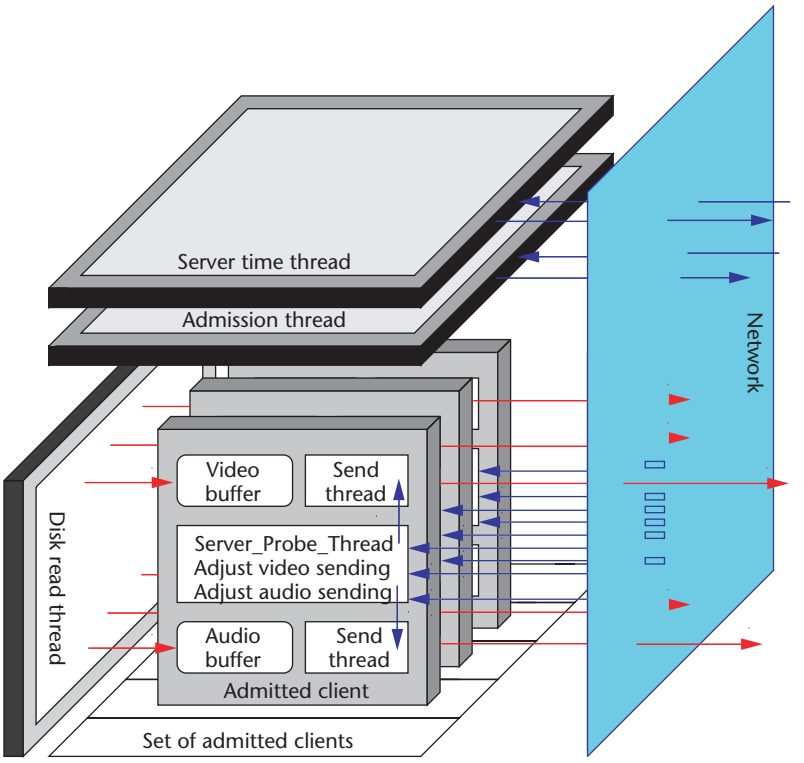

Figure 2. Server design for audio and video streams.

Client design

A client's main functionality is to display multiple media streams to users in the specified format. The client synchronizes the audio and video packets and delivers them to the output devices. In addition, the client sends feedbacks to the server, which the server uses for admission control and scheduling. Figure 3 (next page) shows the client's design. The client back end receives the stream packets from the network and fills in the client caches. The client back end includes two components:

- *Communication module.* The communication module provides the interface to the network layer. It primarily consists of two subsystems to handle user datagram (UDP) and transmission control (TCP) protocols. This lets the client use faster protocols like UDP

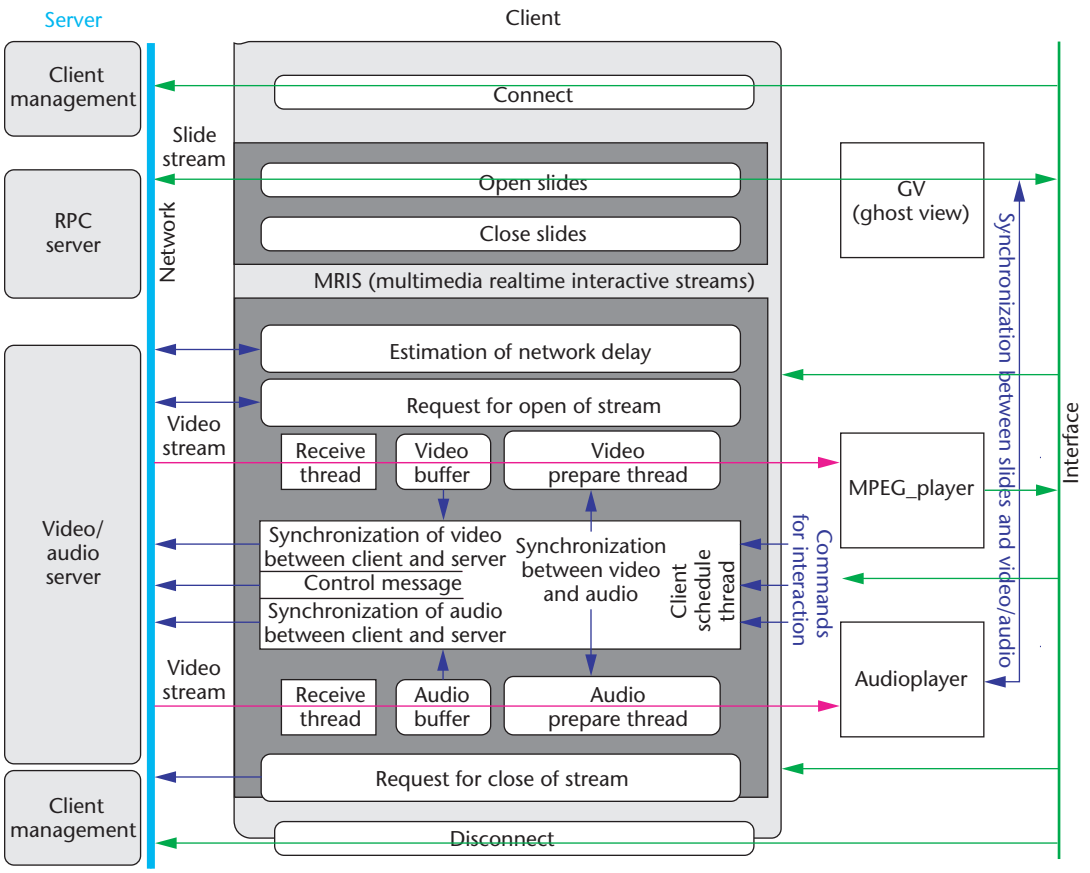

Figure 3. Multiple streams synchronized at the client.

for media streams that can tolerate some data loss and reliable protocols like TCP for feedback messages.

- *Buffer manager*. The buffer manager maintains the client cache in a consistent state.

The client front end reads out the data from the client cache and ensures the synchronized stream presentation. The client front end includes two components:

- *Synchronization manager*. The synchronization manager schedules the multimedia data delivery to the output devices. This module controls delivery of media streams based on QoS parameters that users and the presentation's state define.

- *Presentation manager*. The presentation manager interacts with the GUI and translates user interactions such as **START** and **STOP** into meaningful commands that the other subsystems can understand.

Note that we can retrieve a stream consisting of transparencies (or images) using conventional transmission methods such as remote procedure calls (RPC) and synchronize it with the audio and video streams at the interface level. The advantage of RPC is that standard function calls can implement well-known file functions—such as opening and reading from a certain position—that the server machine then transparently executes. An RPC request may contain commands to find a stream's file length, providing directory content or starting the slide session's delivery. The RPC server retrieves all requested data and sends it back to the client side. Since the scheduling for such data isn't as time critical as audio or video data, the display of such data can tolerate some delay. In our implementation, we synchronize the slides first according to the audio information. If no audio is available, we schedule slides with video; otherwise it behaves like a regular Postscript document.

To make the access to media streams transparent to the application's higher levels, we provide the MultiMedia Real-Time Interactive Stream

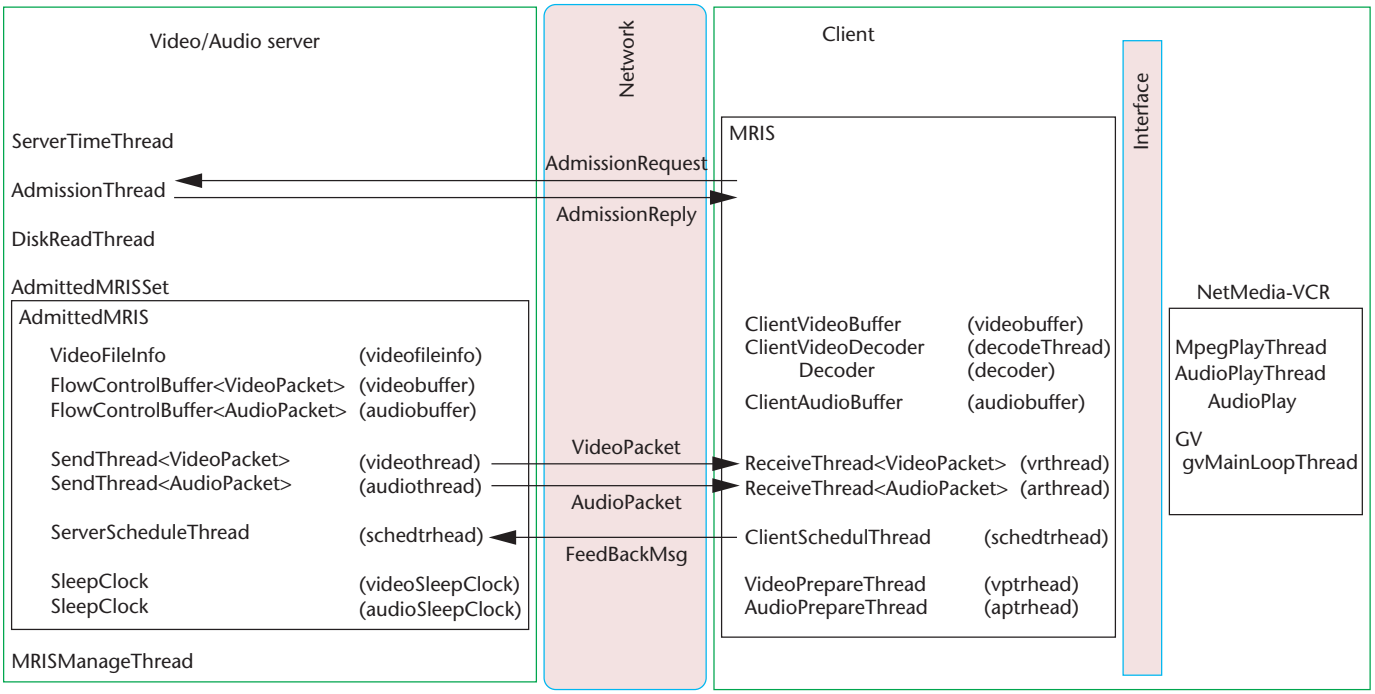

Figure 4. NetMedia class hierarchy using the MRIS.

(MRIS) interface for requesting media stream delivery such as audio, video, or any other media in a synchronized form from the server. MRIS separates the application from the software drivers needed to access multimedia data from remote sites and controls specific audio and video output devices. The MRIS interface has the advantage that the transmission support is independent of the application. Any application that uses synchronized multimedia data can rely on this module. We use the MRIS class to implement our virtual classroom application. Users can synchronize the slides with the video and audio data at the MRIS interface level.

Figure 4 illustrates NetMedia's class hierarchy. The application only contains the transparency-reader Ghostview (GV), control Loop (gvMainLoop), and player units (MpegPlayThread and AudioPlayThread). MRIS handles everything else.

Time control

A crucial design element of a multimedia system is control over the timing requirements of the participating streams so that we can present the streams in a satisfiable fashion. In NetMedia, a robust timing control module (named SleepClock) handles all timing requirements in the system. The novelty of SleepClock is that it can flexibly control and adjust the invocation of periodic events. We use SleepClock to enforce

- end-to-end synchronization between server and client,
- smoothed transmission of streams,
- periodic checks in modules, and
- timing requirements of media streams.

Figure 5 (next page) shows how SleepClock works. Initially, we set the start time of a periodic event sequence and the time interval of each event. We can set SleepClock to sleep until a given event number's appropriate time. Sleeping renders the current process inactive until it reaches the event's time. For example, consider that the system started at time s = 12:00:00 p.m. and the interval is set to 1 minute. If the current process needs to sleep until the number 5 event and the current time is 12:02:30, then a sleep routine will be called with its value set to 2 minutes and 30 seconds. This lets the process sleep until the correct time. If the current time surpasses startime + event_number * interval, then SleepClock will take no action and the process can try to keep up to its timing requirements.

The advantage of SleepClock is that it provides two ways to adjust the progress of events. First, the invocation time of each event can be modified. It does this by assuming a new start time for the event sequence, which results in an acceleration

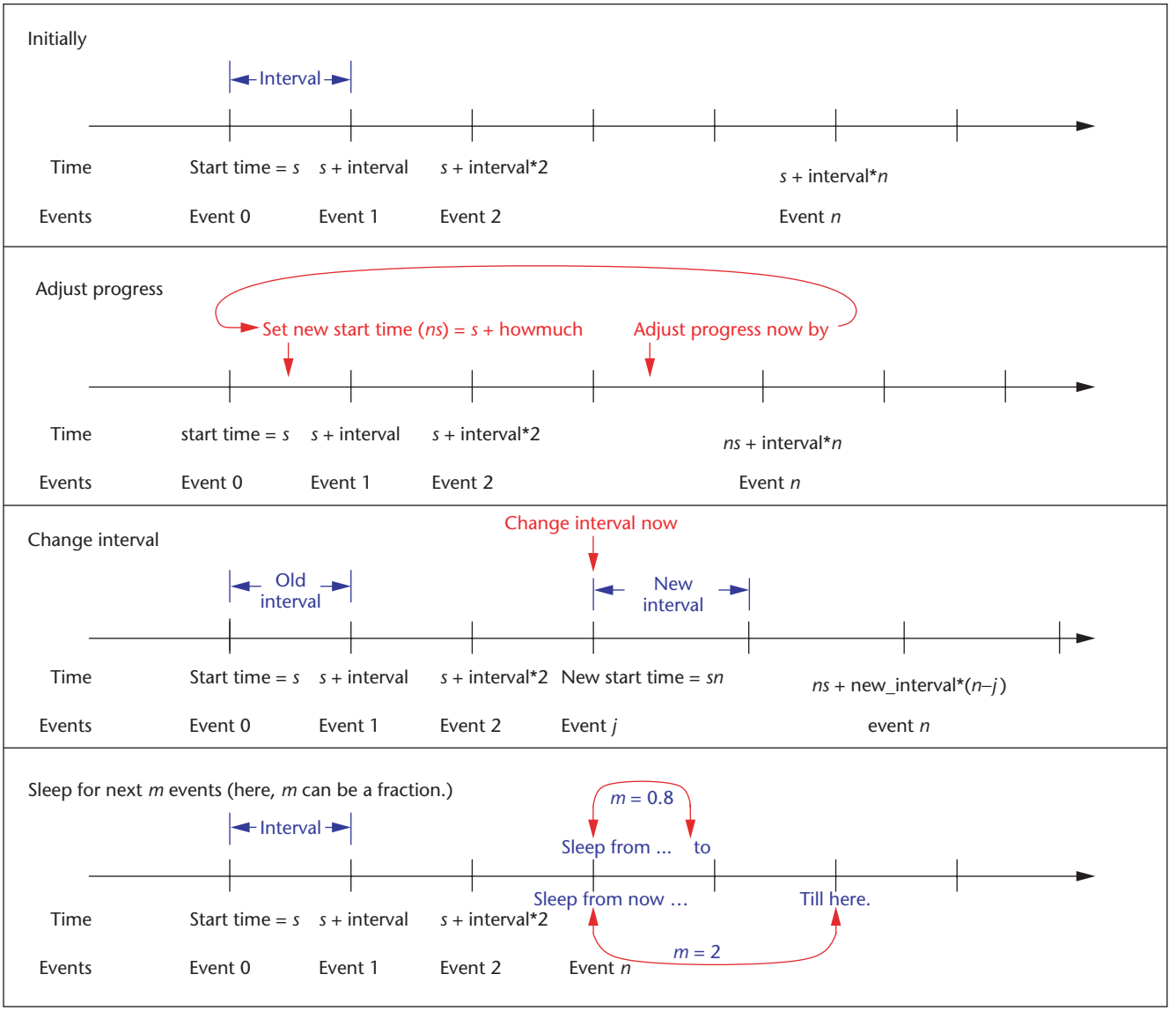

Figure 5. SleepClock design.

or delay for the remaining sequence of events. SleepClock checks the current time in relationship to new_starttime + interval ∗ current_event. For example, consider that the system has streamed out 100 frames when it receives a speedup command. By moving the start time in the past (relative to the old start time), the 101 event (and all following events) will occur sooner in contrast to the original timing schedule. In Figure 5, Adjust Progress shows an example of the modified start time. It achieves the second form of adjustment by modifying the interval between events. Figure 5 gives an example of changing intervals.

Buffering strategies

Buffer management is a crucial part in the NetMedia system, which has two main components: server and client buffers.

Server flow-control buffer

We designed a straightforward server buffer for each stream to perform fast delivery of the stream from the disk to the network. Figure 6 shows the server buffer structure for a stream. A server stream buffer is a circular list of items. Each item may contain one packet. Each server stream buffer has the following characteristics:

- *One producer and one consumer.* The producer writes packets to the buffer and the consumer reads from the buffer.
- *First-in, first-out.* Packets written in first will be read out first.

Specifically, each server stream buffer has two pointers: read pointer (readp) and write pointer (writep). Items from readp to writep are readable; spaces from writep to readp are writeable.

Because a server buffer is designed for data flow control from the disk to the network, the buffer's size can be relatively small. In NetMedia's implementation, the video buffer's size is 100 video packets, where each video packet is 1,024 bytes, and the audio buffer's size is 20 audio packets, where each audio packet is 631 bytes.

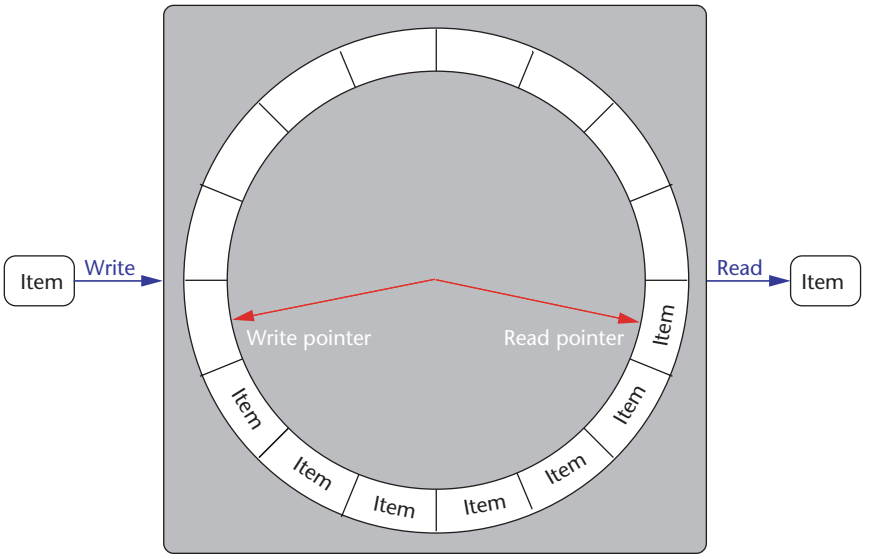

Figure 6. Server flow-control buffer design.

Client packet buffer

Client buffer management is a central part of the NetMedia framework. In contrast to the server buffer that must support fast delivery of the stream from the disk to the network, the purpose of a client buffer is to take care of network delays and out-of-order arriving packets and to efficiently support user interactions.

The client buffer management's novelty is its ability to support interactivity requests, like backward playback or seek. The difficulty experienced with interactive support in video streams is the video encoding used by most advanced compression techniques.[4] Several works propose methods for implementing forward playback but don't adequately deal with backward playback. To solve the decoding problem, we propose an approach[5] to use a separate file for each video stream, preprocessed for backward playback. The advantage is that it doesn't need special client buffer management. The disadvantage is that the system must create a separate file for all video streams, thus increasing the storage used in the system and its cost.

Our approach is to let the client support interactive requests rather than the server. As the client processes video frames, rather than discarding the displayed frames, the client saves the most recent frames to a temporary buffer. We do this to reduce the network's bandwidth requirements in exchange for memory at the client site, because we expect an increase in network load in the next few years compared to falling memory prices.

A circular buffer serves as the client buffer design for interactivity functions. The model proposed here lets all interactive functions occur at any time with an immediate response time. If the buffer is large enough, the interactive request for backward playback can be fed directly out of the buffer instead of requesting a retransmission from the server.

The interactive buffer for a stream has two levels: packet and index buffers. The first level is the packet buffer, which has a set of fixed-size packets. Each packet has one pointer that can point to another packet, so we can easily link any set of packets by pointer manipulation (for example, all packets for a particular frame). The buffer maintains a list of free packet slots.

The second level for audio buffer is the index buffer of a circular list, in which each entry records the packet's playback order. Each packet in an audio stream has a unique number—PacketNo—which indicates the packets' order in the whole stream. For the video buffer, each index in the index buffer points to one frame (which is a list of ordered packets within the frame). We define the point with the minimal packet/frame number as MinIndex and the point where we read data for playback as CurrentIndex. Both indices traverse the circumference in a clockwise manner, as Figure 7 (next page) shows. (Note that in this figure, since the buffer is for a video stream, we named the two pointers MinFrameIndex and CurrentFrameIndex.)

The distance between MinIndex and CurrentIndex is defined as the *history area*. A packet isn't removed immediately after being displayed. It's removed only when some new packet needs space. So valid indices from MinIndex (inclusive) to CurrentIndex (not inclusive) point to the packets or frames that have been displayed and haven't been released yet. (Later the system can use the data to support backward.) To keep

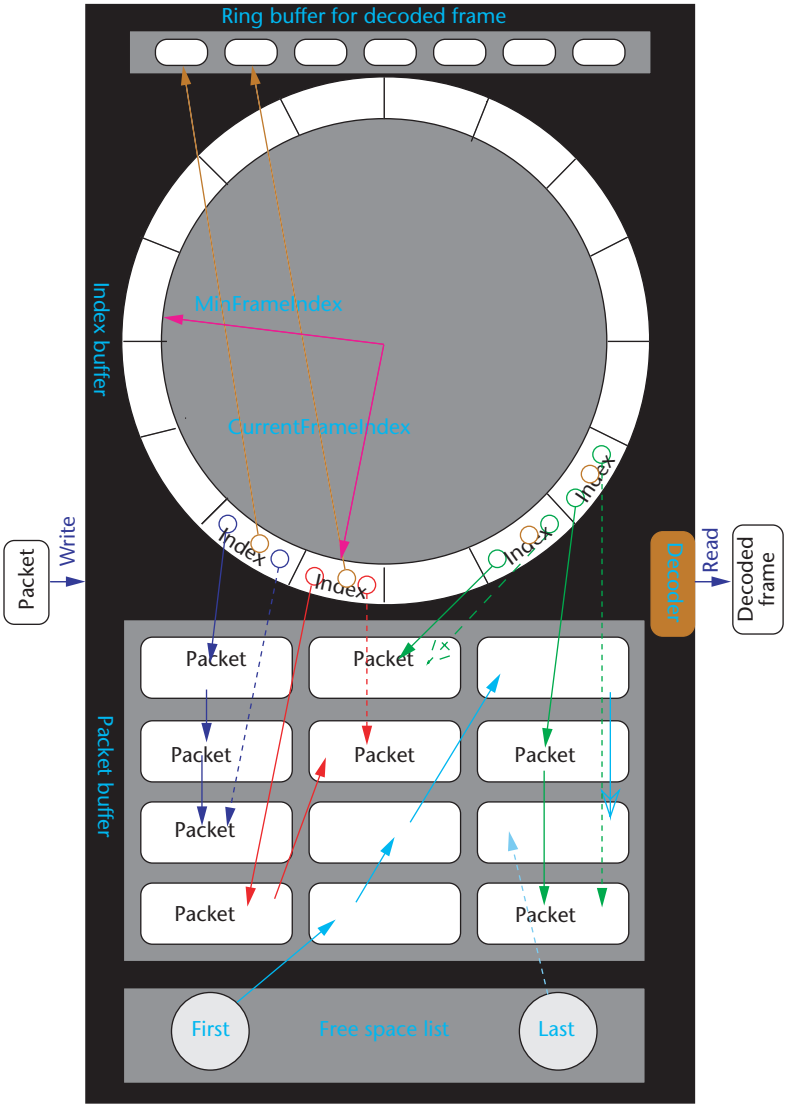

Figure 7. Packet and index buffers for a video stream at the client.

track of the forward playback capabilities, we introduce the packet/frame with the maximum number as MaxIndex. Note that MaxIndex is always between CurrentIndex and MinIndex. We define the distance between CurrentIndex and MaxIndex as the *future area*. Using the history and future areas, users have full interaction capabilities in the two different areas.

When a new packet arrives at the client site, the system checks its packet or frame number (denoted as seqno) and decides if the packet can be stored. We define the valid range of the packet or frame numbers as follows:

$$\text{MinIndex} \le \text{seqno} \le \text{CurrentIndex} + \text{BuffSize} - 1$$

If the packet has a packet or frame number outside this range, then the system considers it a packet loss and discards it. Otherwise, the packet is stored in a packet slot either taken from the free packet slot list or obtained from releasing some history data. Note that once the system releases some history data, it also updates the MinIndex.

To increase the decoding speed of compressed motion predicted frames, we introduce the use of a decode buffer. The MPEG compression scheme[4] introduces interframe dependencies among I, P, and B frame types. To decode a certain frame, previous or future frames might be necessary. For example, given the following frame sequence:

frame type:	I	B	B	P	B	B	P	B	B	I
frame #:	1	2	3	4	5	6	7	8	9	10

frames 1, 4, and 7 must be decoded before frame 9 can be decoded. To speed up the decoding process for the backward playback, we cache the decoded frames within an I-to-I frame sequence. In case of dependencies, we might be able to feed necessary frames out of the decode buffer instead of calling the decoding routine multiple times.

To implement the buffer design and let interactive requests be fed from their local buffer, we need to set a reasonable buffer size for the index and packet buffers. We introduce two consideration factors: Max_Interactive_ Support and Reasonable_Delay. Max_Interactive_Support defines the longest playback time we expect to be buffered in the client for interactive support, thus defining the index buffer's size. Reasonable_Delay defines the delay within which most packets will pass over the network, thus defining the packet buffer's size. In NetMedia's implementation, we defined Reasonable_Delay at 2 seconds. Max_Interactive_Support's size depends on the amount of forward and backward play support. (This can vary from 10 seconds to 5 minutes or more, depending on the memory at the client.)

At the server site, video frames are normally represented in encoding order to make it easier for the decoder to decode in the forward playback. This complicates the decoding of frames for backward playback. We transmit our streams in display order to achieve the minimal distance among the dependent frames for both forward and backward playback decoding. The advantage is that our scheme at the client site has uniform design and the same performance for both forward and backward playback. By using the decoding buffer and our network-control algorithms, we can control the amount of data in the future or backward area

in our buffer. This lets users specify the amount of interactivity support that they want for backward or forward playback without immediate request to the server.

With our buffer design, we can use the history and future areas to feed interactive requests directly from the local buffer. If the request doesn't exceed the cache, the server won't be notified. This will highly reduce the scalability problems faced by the server in case of multiple interactivity requests. The server must handle access outside the history or future area by redirect streaming, but the probability of this case can be drastically reduced by the buffer size amount given at the client.

End-to-end flow control

Various delays and congestion in the network may cause the client buffer underflow or overflow, which may result in discontinuities in stream playback. We propose DSD and PLUS schemes to dynamically monitor the network situation and adapt our system to the current situation of the network. The DSD scheme provides full flow control on stream transmission, and the PLUS scheme uses network status probing and an effective adjustment mechanism to data loss to prevent network congestion. Both schemes work together to find the optimum adjustment considering network delay and data loss rate and ensure no underflow or overflow in the client buffer.

DSD scheme

The basic dynamic end-to-end control scheme used in NetMedia is the DSD algorithm presented in Song et al.[3] For DSD, we define the distance between sending and display at time t as the difference between the nominal display time of the media unit—which is being sent—and the media unit, which is being displayed. Thus, at a given time, let the media unit being displayed have nominal display time T and the media unit being sent have nominal display time t. Then, DSD = $t - T$. Figure 8 illustrates the meaning of DSD in a stream.

The connection between DSD and the network delay is as follows. Comparing it with the network delay, if DSD is too large, then the overflow data loss at the client buffer could be high; if DSD is too small, then the underflow data loss at the client buffer could also be high. Thus, DSD should be large enough to cover the network delay experienced by most media units; it also should be small enough to ensure the client buffer won't overflow.

By adjusting the current DSD in the system to respond to the current network delays, we can monitor the best DSD dynamically. We can then adjust the sending rate to avoid underflow or overflow in the client buffers and alleviate data loss.

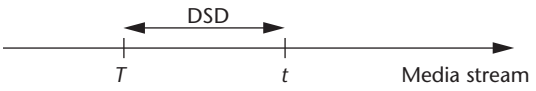

Figure 8. Definition of DSD in a stream.

Initially, we select a DSD based on the up-to-date network load and fix the sending and display rates at the nominal display rate. In the following we show the current DSD calculation, the best DSD, and the DSD adjustment.

Current DSD calculation. At time t, let t_{sent} be the nominal display time of the media unit being sent at the server site and $t_{\text{displayed}}$ be the nominal display time of the media unit being displayed at the client site. By definition, the DSD is

$$\text{DSD}_t = t_{\text{sent}} - t_{\text{displayed}} \quad (1)$$

Suppose at time t, the media unit being displayed has unit number $u^t_{\text{displayed}}$, the media unit being sent has unit number u^t_{sent}, and nominal duration of one unit is UnitDuration. We can then calculate DSD as follows:

$$\text{DSD}_t = (u^t_{\text{sent}} - u^t_{\text{displayed}}) \times \text{UnitDuration} \quad (2)$$

The client can always know $u^t_{\text{displayed}}$ directly, but it can only estimate $u^t_{\text{sent}}$from the information carried by currently arriving media units. Suppose that it sends the current arriving media unit u at time SendingTime_u and its unit number is n_u. Assume that the server is sending with nominal display rate. At time t, we have

$$u^t_{\text{sent}} = n_u + \frac{t - \text{SendingTime}_u}{\text{UnitDuration}} \quad (3)$$

We then obtain

$$\text{DSD} = (n_u - u^t_{\text{displayed}}) \times \text{UnitDuration} + (t - \text{SendingTime}_u) \quad (4)$$

Best DSD calculation. We define an allowable range for DSD:

$$\text{MINDSD} \le \text{DSD} \le \text{MAXDSD},$$

where MINDSD = 0 and MAXDSD = Max_Delay. We then evenly divide interval [MINDSD,

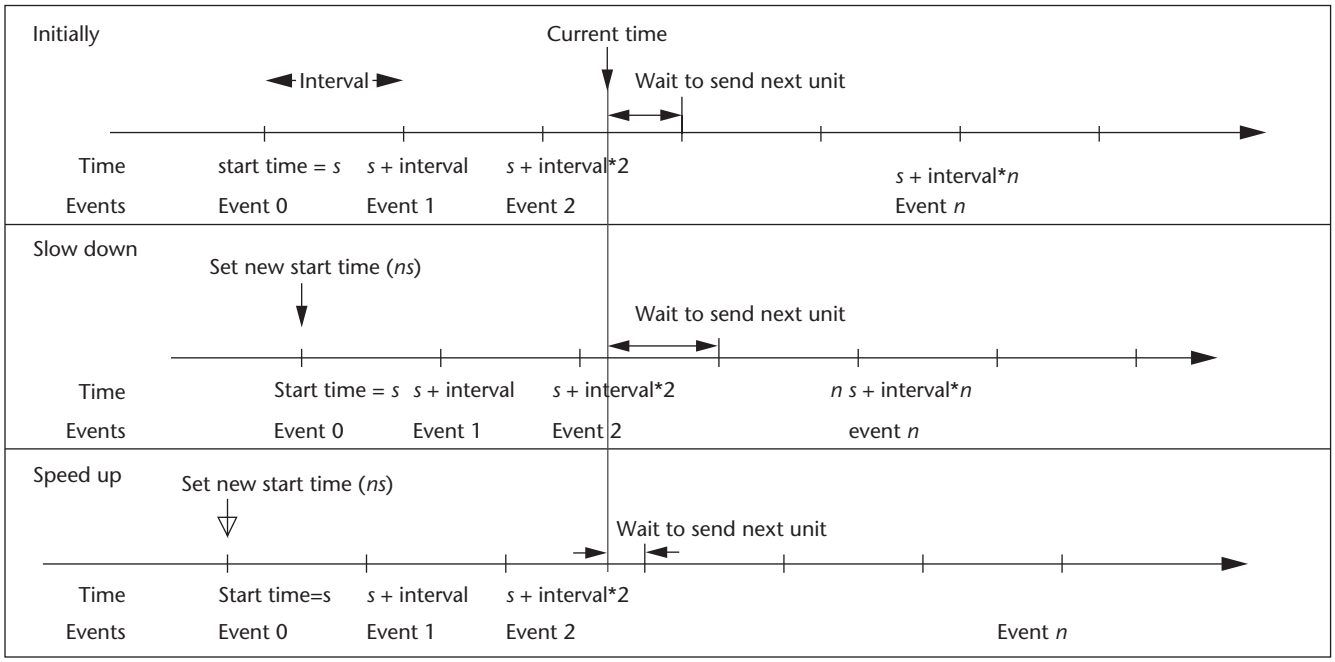

Figure 9. Time control in implementing the DSD scheme.

MAXDSD] into $k - 1$ subintervals with interval points d_1 = MINDSD, d_2, d_3, … d_k = MAXDSD. Here k can be 10, 20, or even 100. The tradeoff is that if k is too small, the best DSD found might not be good enough; if k is too big, the overhead in calculating best DSD is too large. For each d_i, we keep track of data loss with respect to it.

Here's the definition of relative data loss with respect to d_i. Let d be the current DSD. Suppose we have a virtual client, which is the same as the actual client, with the same buffer size and the same displaying time for any media unit. The difference is that, suppose for the real client, it sends a real media unit at time t. It then should be displayed at time $t + d$. For the virtual client, this unit is sent at time t and displayed at time $t + d_i$. So the virtual *DSD* for this unit is d_i. Upon this assumption, we define the virtual client's loss rate as the relative loss rate with respect to d_i. When a media unit arrives, virtual clients first check if the unit is late. Suppose the current time is currentT, the virtual client's DSD is d, and the sending time of this media unit is sendT. If sendT + d < currentT, then the unit is late. If the unit is late, it's counted as a relative loss for this virtual client.

The best DSD finder keeps track of these virtual clients. It launches the best DSD calculation once in a fixed time period (200 milliseconds in NetMedia's implementation). When it needs a new (best) DSD, the best DSD finder browses virtual clients and reports the DSD in the virtual client with minimum loss rate as the best DSD. Note that DSD only captures relative packet loss, which means that the packet arrived at the client site but was too late for display. The PLUS protocol captures any data loss due to network congestion.

Adjustment to network delay. The system chooses a DSD as an application begins, based on the current network load. If the best DSD differs from the current DSD, we can adjust it in two ways. One is to change the sending rate and another is to change the display rate. We can adjust the display rate by the timing routine at the client site. We only adjust video streams.

At any calculation point, if the client calls for adjustment, then it sends a feedback message to the server side to adjust the sending rate, so that it changes the system DSD to the best DSD. The feedback messages will carry the Δd—the difference of the best DSD and the current DSD. A negative Δd means the sending rate should slow down and a positive Δd means the sending rate should speed up. The feedback data also contains information about the loss rate to initiate frame dropping at the server site if necessary.

To slow down, the server stops sending for $|\Delta d|$ time. To speed up, the server always assumes that the current display rate is at nominal display rate r. Let the maximum available sending rate over the network be R. The server will speed up using the maximum data rate for a time period. After the reaction time finishes, the server will use the nominal display rate r as the sending rate.

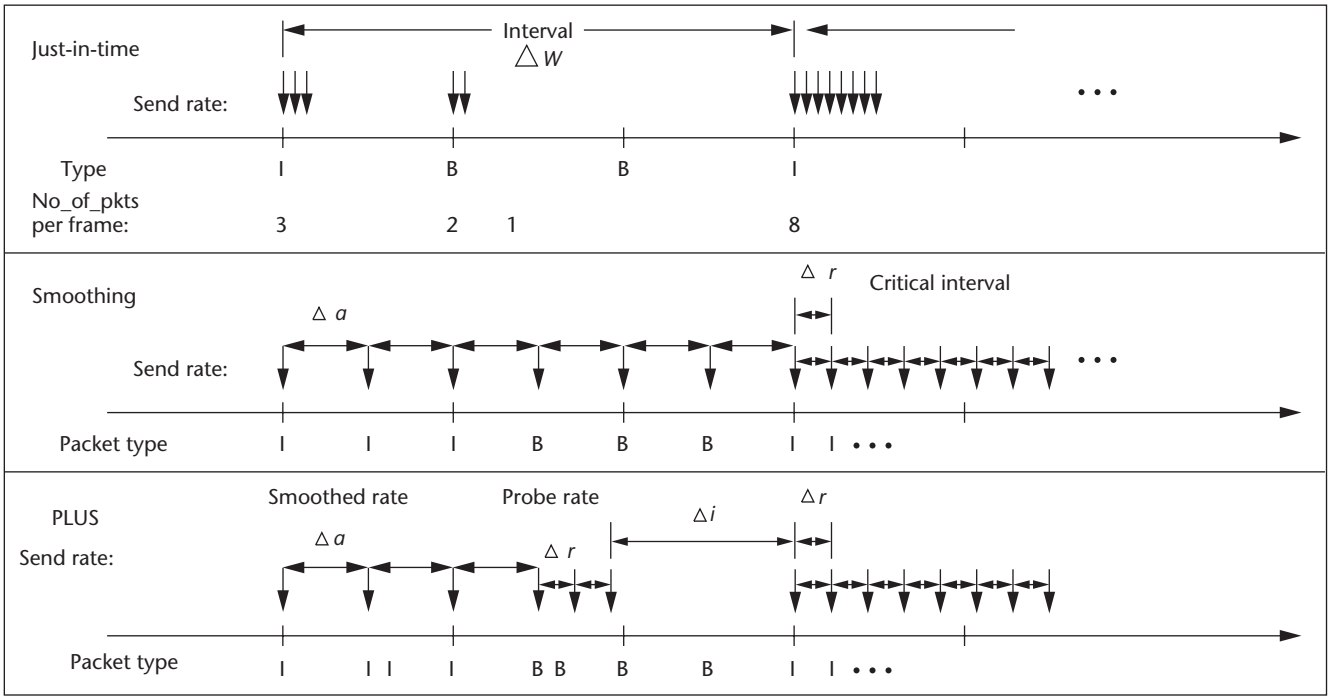

Figure 10. PLUS probing scheme.

Figure 9 shows SleepClock's use in the DSD scheme. The scheme adjusts the start time to change the stream's sending times in the following events, which either increases or decreases the sending bandwidth temporarily. This can adjust the stream for varying end-to-end delays in the network.

PLUS scheme

The DSD scheme doesn't address the congestion problem in the network. It also doesn't consider the effect of data compression, which may introduce burstiness into the data streams. The burstiness results in different bandwidth requirements during the stream's transmission. This makes it difficult to come up with a resource and adaptation scheme because the bandwidth requirements always change. To address these issues, we developed PLUS as a new flow and congestion control scheme for distributed multimedia presentation systems. This scheme uses network situation probing and an effective adjustment mechanism to data loss to prevent network congestion. We also designed the proposed scheme to scale with increasing numbers of PLUS-based streaming traffic and to live in harmony with TCP-based traffic. With the PLUS protocol we address how to avoid congestion rather than react to it with the probing scheme's help.

The PLUS protocol eases bandwidth fluctuations by grouping together some media units (frames for video) in a window interval ΔW (Figure 10). We define easing bandwidth in a given window as *smoothing*.[6] One way of smoothing is to send out the data at the average bandwidth requirement for the window (Equation 5):

$$\Delta_a = \frac{\Delta W}{\text{number\_of\_packets\_in\_current\_interval}} \quad (5)$$

By using this method, the client can guarantee that the bandwidth needed is minimal and constant through the interval. The disadvantage of this method is that the feedback received only applies to the current or past bandwidth requirements. It doesn't consider that there might be a higher bandwidth request in the future, which the network may not be able to process.

For each interval ΔW, we identify a critical interval. The critical interval is an interval in the future playback time that contains the maximum number of packets. The aim of the PLUS probing scheme is to test if the network can support the critical interval. We set ΔW to be a 5-second smoothing window in NetMedia's implementation. This is a reasonable time to reduce the bandwidth through smoothing. It's also granular enough to detect sequences with fast movements (which result in higher numbers of packets per frame and therefore provide the bottleneck bandwidth of a stream).

Once we determine the critical interval for each sequence, we apply our smoothing and probing scheme. The critical bandwidth in the

future, at a given interval, is provided by its critical interval. To find the minimal bandwidth requirement for the critical interval, we apply the averaging algorithm, which spreads the constant-sized packets evenly. This leads to a sending difference between consecutive packets, which we define by

$$\Delta r = \frac{\Delta W}{\text{pkts\_in\_critical\_interval}} \quad (6)$$

According to Keshev,[7] the packet pair approach at the receiver site can estimate a connection's bottleneck bandwidth. The essential idea is that the interpacket spacing of two packets will be proportional to the time required for the bottleneck router to process the second packet. We calculate the bottleneck bandwidth as

$$b = \frac{\text{packet\_size}}{\Delta r} \quad (7)$$

In MPEG-encoded streams the loss of I or P frames results in further dependent frames loss. Only the loss of B frames doesn't result in further loss. A B frame loss (in presentation) results only in a short degradation of the quality in playback. We use B frames to probe the network to see if it can support the critical interval and thereby protect critical frame loss like I or P frames (as the lower part of Figure 10 shows).

Instead of spreading each packet evenly within our 5-second smoothing window, we use the bottleneck bandwidth for sending out all B frames. Critical packets (belonging to I or P) will still be sent out with the average bandwidth calculated in Equation 5. Our scheme will thereby punctually probe the network while acknowledgments will give direct feedback ahead of time if the network can support the critical bandwidth. In case of congestion, we'll initiate a multiplicative backoff of the sending rate. B frames have less chance of survival in our scheme in case of congestion. This isn't a disadvantage because we can proactively provide a bandwidth reduction at the server site by dropping noncritical B frames in time to increase the survival rate of critical packets (thereby increasing the subsequent B frames' survival number).

Concurrent probing of different PLUS streams reports a conservative estimation of the current network situation. We based the estimation on the bandwidth need of the streams' critical intervals and not necessarily the current average interval. This behavior lets the PLUS stream be aware of its surrounding and responsive to protect critical packets when reaching the connection's maximum capacity. If the concurrent probing causes packet loss, PLUS streams back off, letting them live in harmony with TCP and other PLUS streams. Sequential probing of different PLUS streams could report an estimation that leads to data loss in case multiple PLUS streams send their critical interval at exactly the same time, each assuming the network can support it. The probability of such a situation is low and we can reduce it further with the number of B frames in a stream.

We can also use SleepClock to sleep for the next m events. Note that m can be a fraction of an event. We use this to implement smoothed stream transmission in the PLUS protocol. A video stream frame consists of a sequence of one or more packets. Each frame has to be sent out according to its nominal display rate. However, I frames normally contain a larger number of packets than P or B frames, which must be sent per interval. To reduce the bandwidth required by I frames, we can smooth packets over a number of n frames (which equals n events). To uniformly send the packets, we can assign a fraction of the interval time for an event to each packet.

Integration of the DSD and PLUS schemes

In addition to adjusting to the best DSD, the DSD scheme also responds to the adjustment needed by the PLUS scheme. It's straightforward to respond to slow-down requests. In case of speedup, the time period t of the response at the server side must be carefully calculated. Assume that the nominal display rate is r, which is smoothed based on the PLUS scheme, and the current maximum transmission rate available is R ($R > r$). We then must maintain

$$R * t - r * t \leq B_{avl}$$

where B_{avl} is the extra buffer space available beyond the data buffered for the nominal playback. Thus,

$$t \leq \frac{B_{avl}}{R - r}$$

During the congestion period, instead of increasing the DSD period with a large amount of data to transmit, the system drops noncritical B frames to maintain a reasonable DSD.

Playout synchronization

Consider the issue of enforcing smooth multimedia stream presentation at client sites. This is critical for building an adaptive presentation system that can handle high rate stream presentation

variance and delay variance of networks. In Johnson and Zhang,[8] we observed that dynamic playout scheduling can greatly enhance the smoothness of media-stream presentations. We thus formulated a framework for various presentation scheduling algorithms to support QoS requirements specified in the presentations.[8]

We defined various QoS parameters to specify requirements on multiple-stream presentations.[9] At the client, users submit their QoS parameters for processing their multimedia data. For example, Little and Ghafoor[9] have proposed QoS parameters including average delay, speed ratio, utilization, jitter, and skew to measure the QoS for multimedia data presentation. In our playout scheduling framework, we consider individual stream (intrastream constraints) QoS parameters:

- *Maximum rate change* denotes the range within which the rendition rate of stream *s* can vary. This is a fraction of the nominal rate for *s*.

- *Maximum instantaneous drift* denotes the maximum time difference at any given time between the presentation progress of the stream and the presentation progress at the nominal presentation rate.

We also consider QOS parameters for multiple streams (interstream constraint):

- *Synchronization jitter* denotes the maximum allowable drift between the constituent presentation streams. This drift is the difference between the presentation progress of the fastest and slowest streams.

We designed a scheduler within the NetMedia-client to support smooth multimedia stream presentations at the client site in the distributed environment. Our primary goal in the design of playout scheduling algorithms is to create smooth and relatively hiccup-free presentations in the delay-prone environments.

Our algorithms can dynamically adjust any rate variations that aren't maintained in the data transmission. We define the *rendition rate* as the instantaneous rate of a media stream's presentation. Each stream must report to the scheduler (which implements the synchronization algorithm) its own progress periodically. The scheduler in turn reports to each stream the rendition rate required to maintain the desired presentation. The individual streams must try to follow this rendition rate. The client has several threads running concurrently to achieve the presentation.

Let the nominal rendition rate for a stream s_i be denoted by R^n_{si}. We express this rate as the number of stream granules per unit time. For example, the nominal rendition rate for video could be 30 frames per second, with one frame comprising a granule. To meet this scheduling requirement, the time period between consecutive frames must be 1/30 seconds. Thus, we can achieve intrastream synchronization by displaying the frames at this rate. However, because of various factors affecting system load, it may not always be possible to achieve this rate. We must therefore specify a reasonable range within which display operations will fall. We integrate two intrastream QoS parameters—maximum rate change c_{si} and maximum instantaneous drift d_{si}—and one interstream QoS parameter, synchronization jitter *j*, into the stream presentation.

We maintain intrastream constraints by enforcing constraints imposed by c_{si} and d_{si}. The maximum rate change c_{si} represents the range within which the rendition rate of stream s_i can vary. It's a fraction of R^n_{si},

$$R^n_{s_i} * (1 - c_{s_i}) \le R_{s_i}(t) \le R^n_{s_i} * (1 + c_{s_i}), \quad (8)$$

where $R_{si}(t)$ is the rendition rate of s_i at a time instant *t*. We assume that when we display a stream granule, the rendition rate $R_{si}(t)$ of s_i remains unchanged.

We now define *progress* $p_{si}(t)$ of a stream s_i at time *t* as the presentation time of granules displayed to date, assuming the nominal presentation rate. In general, we measure progress in terms of the granules displayed to date. Let the presentation of stream s_i start at time t_0. We say that stream s_i is progressing at the nominal rate if, at time *t*, the granules displayed to date are $R^n_{si} * (t - t_0)$. In our discussion, we specify progress in terms of time rather than in granules. Stream s_i is progressing at the nominal rate if, at time *t*, the progress measured in time is also $t - t_0$. We arrive at this by dividing the granules displayed to date ($R^n_{si} * (t - t_0)$ for the nominal rate) by R^n_{si}. We can express the progress in terms of time by dividing the granules displayed by R^n_{si}. If the progress is either less or greater than $t - t_0$, then the presentation is either below or above the nominal rate. The presentation is at the nominal rate if the progress equals $t - t_0$.

The maximum instantaneous drift at any given time d_{si} denotes the maximum time difference

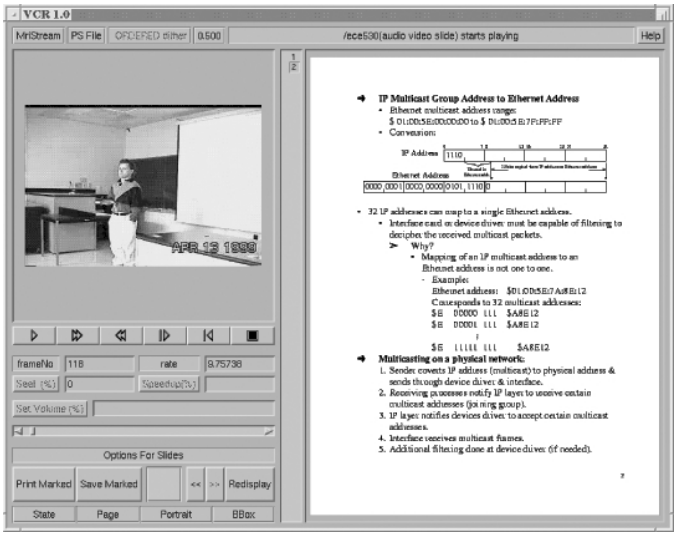

Figure 11. Screen shot of the NetMedia-VCR user interface.

between the progress of the stream and the progress assumed under the nominal presentation rate. If the stream started at time t_0, then, given the nominal presentation rate, the progress at time t is $t - t_0$. Thus, we should have

$$| p_{s_i}(t) - (t - t_0) | \leq d_{s_i} \quad (9)$$

We assume that users can specify an interstream jitter j to enforce synchronization requirements. The interstream synchronization requirements can be met if, for each synchronization point at time t and all streams s_i and s_k in the presentation,

$$| p_{s_i}(t) - p_{s_k}(t) | \leq j \quad (10)$$

The scheduling algorithms presented in Johnson and Zhang[8] dynamically formulate rendition rates for media streams such that the three conditions given in Equations 8, 9, and 10 are satisfied. The central idea in these algorithms is that at any synchronization point, the rates of all streams from this point to the next synchronization point are calculated to meet the constraints. The intrastream constraints give us the allowed range for each stream rate. The interstream constraint gives the relationship among these rates. The algorithms give two options—minimizing change in rate or minimizing jitter. You can find the details of these algorithms in Johnson and Zhang.[8]

For synchronization between video and audio streams, we use the playout scheduling algorithms proposed in Johnson and Zhang.[8] When asynchrony occurs between audio and video streams, it changes the video assembling rate (hence, the video display rate) while maintaining the audio assembling rate (hence, the audio display rate) as the nominal audio display rate. We change the video assembling rate by changing SleepClock's interval.

Experiments

We've developed a multimedia presentation application prototype called NetMedia Virtual Classroom (termed NetMedia-VCR) based on the NetMedia design. We aim to provide a distributed learning environment for students. We implemented middleware architecture and prototype in C++ for Sun machines, which the campus widely uses. As part of the preprocessing work, we recorded and provided lectures in a database at the departmental server. The idea is that students can remotely connect to the server and retrieve lecture material for review and quiz preparation.

Figure 11 shows the graphical user interface of NetMedia-VCR. Students can customize the presentation to fit their individual learning requirements. As Figure 11 shows, students can construct a presentation out of audio, video, and transparencies. Students have free access to skip, seek, rewind, and fast forward learning sessions. They can also mark transparencies and print them for later use.

Table 1. CSE530 test stream and its burstiness.

Course Title	Total Size (Mbytes)	Encoding Type	Resolution	Average Size (Bytes)	Largest Frame	Smallest Frame
CSE530.v	17.9	MPEG-1	320 to 240	3647	9846	1001
CSE530.a	2.4	Raw audio	–	556	–*	–*
CSE530.s	0.2	Postscript file	–	–	–	–

* The audio size is constant per time unit.

– The information doesn't apply.

Users can start the server on any Sun-OS compatible machine. We tested the system at four different sites. We ran the server in Buffalo, New York, Purdue University, Middle East Technical University, Turkey, and the University of Technology, Darmstadt, Germany. The server requires about 0.15 percent system resources on a Sun Ultra 10 with 128 Mbytes, delivering audio, video, and transparencies. The server uses 2.2 Mbytes of disk space. For transparency data delivery, we start an additional remote procedure call (RPC) server. During the tests, the client resided in our Buffalo lab. We used a Sun Ultra 5 with 64 Mbytes of memory as the client.

Researchers have defined various QoS parameters to specify QoS requirements on multiple-stream presentations.[9-11] In our experiments, we measured two particularly important QoS parameters—latency and jitter. During each experiment we recorded the jitter, network delay, and packet loss. We used multiple recordings from lectures provided at the State University of New York at Buffalo as test streams. Table 1 presents the statistics about a stream from our CSE530 Network Communication class lectures.

We conducted test runs every hour to all four sites over a three-month period. We determined the network performance at these sites by the average packet loss experienced during the experiments (Buffalo's average packet loss was 0.01 percent, Purdue's was 1.3 percent, and Germany's was 8.34 percent). Network connections at the Buffalo campus were nearly 100 percent loss-free. All connections ran at 100baseT supported by a Fiber Distributed Data Interface (FDDI) backbone passing at most one router. We classified the network load during the experiments as ideal. Experiments to Purdue and Germany experienced light packet loss most of the time. Packets normally traveled through four (Purdue) to nine (Germany) routers.

Figure 12 shows the jitter between audio and video during the test runs. Measured using the synchronization algorithm given in Johnson and Zhang,[8] the jitter obtained on the Buffalo campus remains on average in the ranges of 0 to 60 milliseconds (ms) throughout the presentation, which is much less than the upper bound (80 ms) given in Steinmetz and Nahrstedt.[10] Jitter obtained from Purdue and Germany experienced spikes in case of packet loss. On average, the jitter was still below the requested 80 ms.

Figure 13 (next page) shows loss rates for audio and video streams with respect to different DSDs calculated by the BestDSD Finder. From these two figures, we can observe that, at any time, the DSDs with the lowest data loss rate (0 in this case) form a range. During the entire presentation, all the

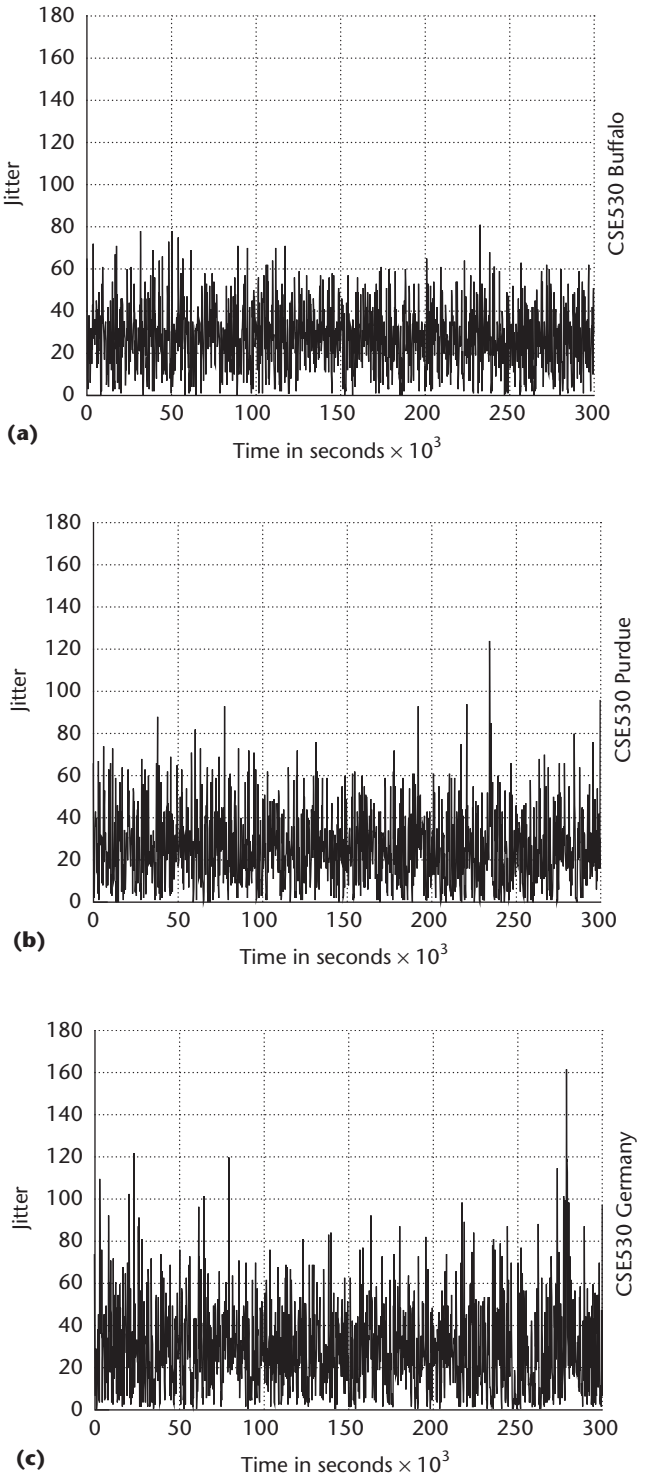

Figure 12. Jitter between audio and video: servers at (a) Buffalo, (b) Purdue, and (c) Germany.

ranges form a valley. Then the finder chooses the best DSDs to form a line within this valley.

Figure 14a compares the average loss recorded daily from the PLUS protocol against a 5-second smoothing implementation with the server in Middle East Technical University, Turkey. The PLUS protocol could reduce the loss rate mainly because of the bandwidth reduction in case of congestion. To test the effect of proactively testing the available bandwidth of the PLUS protocol, we compared the number of received and successfully decoded frames (Figure 14b) between the smoothed transmission and the PLUS scheme.

On average, the PLUS scheme could deliver 47 percent more frames to the client than the smoothed transmission. On one hand, we can achieve this by backing off the sending rate in congestion detection. To study the effect on proactively testing the network, we compared the PLUS protocol to a smoothed transmission, which backs off if the packet loss exceeds 10 percent within a 5-second test interval. As Figure 14b shows, the PLUS protocol doesn't rely on just the backoff mechanism to protect its content. The increased number of saved B frames (even though they send more aggressively) could be achieved because it saves a larger number of critical frames. The probing nature of the PLUS protocol causes this. Transmitting B frames with a higher bandwidth causes the router to be more likely to drop this kind of frame (and initiate proactive frame dropping). Probing also pauses before a critical packet stream (thereby giving the router a chance to switch to a different stream and process the critical packets later), which positively impacts the survival rate of critical packets.

The experiments clearly demonstrate our framework's performance and usability. Interactive requests could be delivered immediately during any test run. As long as the network isn't

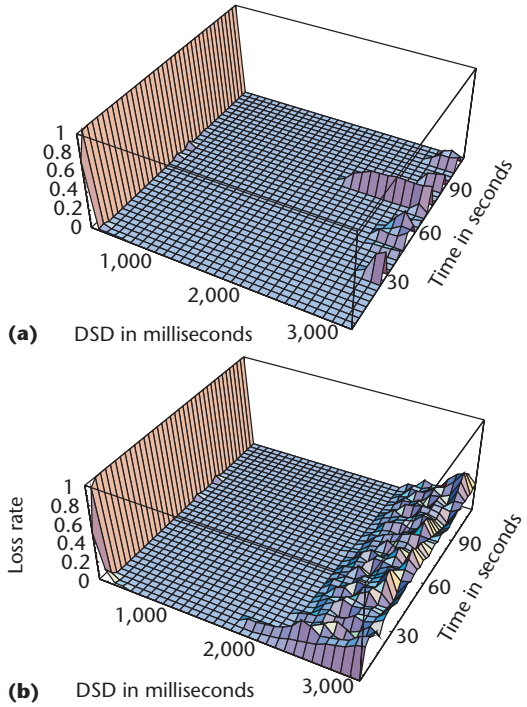

Figure 13. Loss rates at different DSDs with the server in Germany: (a) video stream and (b) audio stream.

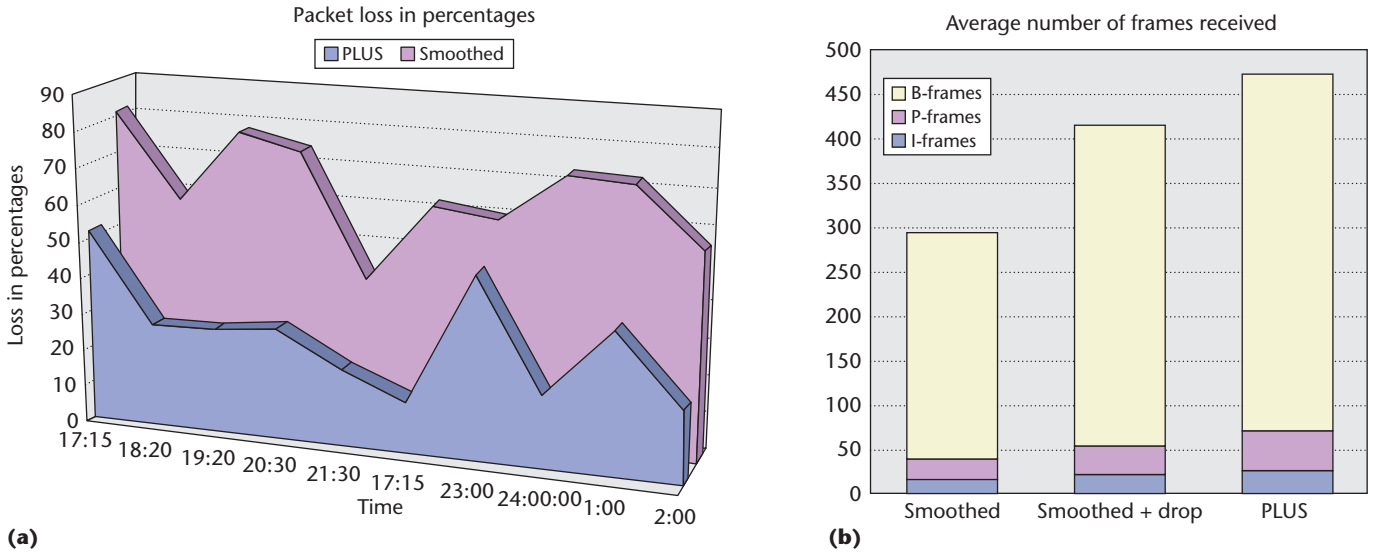

Figure 14. (a) Comparison of the PLUS protocol against 5-second smoothing with the server in Turkey. (b) Comparison of received and decoded frames with the server in Turkey.

completely cloaked we can guarantee the availability and timeliness of data as demanded.

Conclusion

The advantage of the middleware NetMedia is to give application developers a set of services that hides the complexity of distributed multimedia data retrieval, transmission, and synchronization. From the point of the application developer, access to multimedia data becomes as easy as reading a file from the local disk. To achieve this, the system has to integrate connection management, buffer management, payload delivery, QoS management, and synchronization into one framework and provide an easily accessible application programming interface (API) to the developer. The NetMedia framework is a unicast middleware. One future aim is the integration of multicasting streams into our proposed NetMedia middleware.

MM

References

1. T. Lee et al., "Querying Multimedia Presentations Based on Content," *IEEE Trans. Knowledge and Data Eng.*, vol. 11, no. 3, May/June 1999, pp. 361-385.
2. E. Fox and L. Kieffer, "Multimedia Curricula, Courses, and Knowledge Modules," *ACM Computing Surveys*, vol. 27, no. 4, Dec. 1995, pp. 549-551.
3. Y. Song, M. Mielke, and A. Zhang, "NetMedia: Synchronized Streaming of Multimedia Presentations in Distributed Environments," *Proc. IEEE Int'l Conf. Multimedia Computing and Systems*, IEEE CS Press, Los Alamitos, Calif., 1999, pp. 585-590.
4. B. Haskell, A. Puri, and A. Netravaldi, *Digital Video: An Introduction to MPEG2*, Chapman and Hall, New York, 1996.
5. M. Vernick, *The Design, Implementation, and Analysis of the Stony Brook Video Server*, PhD thesis, Dept. Computer Science, SUNY Stony Brook, New York, 1996.
6. W. Feng, F. Jahanian, and S. Sechrest, "Providing VCR Functionality in a Constant Quality Video On-Demand Transportation Service," *Proc. IEEE Multimedia*, IEEE CS Press, Los Alamitos, Calif., 1996, pp. 127-135.
7. S. Keshav, "A Control-Theoretic Approach to Flow Control," *Proc. Conf. Comm. Architecture and Protocols*, ACM Press, New York, 1991, pp. 3-15.
8. T.V. Johnson and A. Zhang, "Dynamic Playout Scheduling Algorithms for Continuous Multimedia Streams," *ACM Multimedia Systems*, vol. 7, no. 4, July 1999, pp. 312-325.
9. T. Little and A. Ghafoor, "Network Considerations for Distributed Multimedia Object Composition and Communication," *IEEE Network*, vol. 4, no. 6, Nov. 1990, pp. 32-49.
10. R. Steinmetz and K. Nahrstedt, "Multimedia: Computing," *Communications and Applications*, Prentice Hall, New York, 1995.
11. D. Wijesekera and J. Srivastava, "Quality of Server (QoS) Metrics for Continuous Media," *Int'l J. Multimedia Tools and Applications*, vol. 3, no. 2, Sept. 1996, pp. 127-166.

Aidong Zhang is an associate professor in the Department of Computer Science and Engineering at State University of New York at Buffalo. She received her PhD in computer science from Purdue University in 1994. Her research interests include content-based image retrieval, digital libraries, and data mining. She is co-chair of the technical program committee for ACM Multimedia 2001 and a NSF Career award recipient.

Yuqing Song received his BS in mathematics from Nanjing University in China in 1990, and his MS in computer science in 1993 from the Institute of Mathematics, Chinese Academy of Sciences. He expects to complete his PhD in computer science from the State University of New York at Buffalo in May 2002. His research interests are multimedia, image processing, and computer vision.

Markus Mielke is a program manager with the Internet Explorer team at Microsoft Corporation. His research interests include multimedia systems, streaming technologies, and peer-to-peer and Web services. He earned his MS degree in computer engineering from the University of Technology, Darmstadt, Germany. He received his PhD in computer science and engineering from the State University of New York at Buffalo in 2000.

Readers may contact Aidong Zhang at the Dept. of Computer Science and Engineering, State Univ. of New York, Buffalo, NY 14260, email azhang@cse.buffalo.edu.