An Experimental Delay Analysis Based on M/G/1-Vacation Queues for Local Audio/Video Streams

Doo-Hyun Kim, Kyung Hee Lee, Sang Hwan Kung, and Jin Hyung Kim

CONTENTS

ABSTRACT

The delay which is one of the quality of service parameters is considered to be a crucial factor for the effective usage of real-time audio and video streams in interactive multimedia collaborations. Among the various causes of the delay, we focus in this paper on the local delay concerned with the schemes which handle continuous inflow of encoded data from constant or variable bit-rate audio and video encoders. We introduce two kinds of implementation approaches, pull model and push model. While the pull model periodically pumps out the incoming data from the system buffer, the push model receives events from the device drivers. From our experiments based on Windows NT 3.51, it is shown that the push model outperforms the other for both constant and variable bit-rate streams in terms of the local delay, when the system suffers reasonable loads. We interpret this experimental data with M/G/1 multiple vacation queuing theories, and show that it is consistent with the queuing theoretic interpretations.

I. INTRODUCTION

1. Motivation

We have developed a prototype distributed multimedia stream processing server, MuX-II [1], [4], [5] which provides intramedia and intermedia synchronizations for networked real-time audio and video streams on Window NT 3.51 and Solaris 4.3. The MuX-II is aimed to support realistic conversations among end users participating a teleconference.

In order to achieve this goal, a variety of quality of service (QoS) parameters [18], [19] has to be optimized in many points such as the coder/decoder (CODEC), network bandwidth allocations, local resource scheduling, and so on. Among the various QoS parameters at such points, we focus in this paper on the local delay concerned with the schemes for processing continuous inflow of real-time compressed data from the audio and video encoder. Examples of such encoders include pulse code modulation (PCM) [2] and H.261 [3].

We investigate two implementation schemes, pull model and push model, for this local input processing. While the pull model periodically pumps the incoming data out from the system buffer, the push model receives events sent from device drivers for encoding hardware. The pull model is more meaningful in point of operating system views since it only requires a periodic resource scheduling which is heavily studied in real-time task scheduling literature [11]. But the pull model necessarily suffers delay of the half of the period in average, while the push model does not have such intrinsic handicaps. So it is necessary to compare each other in diverse situations in terms of offered load, stochastic characteristics, e.g., mean and variances, of the arrival data and their processing time.

2. Related Works

Since the pull model has many attractive points in the view of resource scheduling, much of the recent literature has been focused on periodic workload models [11], [12]. As a principal advantage of these models, QoS values can be easily calculated using the results from the field of real-time scheduling. More-over, these models basically require the specification of only maximum packet rate and size, which can be easily derived from the application, regardless of either variable bit-rate (VBR) or constant bit-rate (CBR) streams.

The non-periodic traffic model which has been a traditional basis for the studies of VBR streams assumes that the over-reservation of resource capacities which frequently happens in the periodic workload models leads to an underusage of the system and, finally, to the needless rejection of new reservation requests. This property of the periodic traffic model has led other researchers to the conclusion that it is not adequate for handling VBR streams [13], [14]. They prefer somewhat complex approaches that can model the traffic behavior more

accurately and thus lead to less wasteful resource reservation.

While there have been much discussions for traffic estimation and optimization, International Telecommunication Union-Telecommunications (ITU-T) recently approved recommendation H.323 aimed for supporting QoS non-guaranteed LAN-based multiparty audio-video teleconferences [23]. This recommendation includes several specifications for variable bitrate audio-video CODECs such as H.261, H.263, and ADPCM. As an alternative approach for LAN or Internet-based multimedia applications, Multimedia Communication Forum (MMCF) [8], [10], [18] is working on defining a reference architecture [8] for distributed multimedia platforms. This architecture defines domains and their application program interfaces (APIs) with object-oriented concepts. The domains include applications, middleware, media device interface (MDI) [25], transport service interface (TSI), QoS Management, and so on. As applications, multimedia desktop collaboration (MDC), multimedia mail and message (MMM), and multimedia information retrieval (MIR) are under discussion. Among those domains, MDI is a device abstraction layer to support device-independent interfaces for diverse local multimedia devices so that the middleware and application programs should not be reprogrammed or reorganized by any change of local devices. Devices are abstracted and treated as objects according to certain class hierarchies. To publicize any objects abstracted by MDI, the middleware [10] provides naming and trading services in the distributed environments. The primary purpose of MuX-II server is to provide generic but optimized local stream connections between local multimedia devices and multicast transport facilities as an essential stream-processing component for on-going standardization activities like MMCF MDI as well as ITU-T standard audiovisual teleconferences.

3. Our Approach

We understand that the above discussions have been done in terms of the network delay. Compared to these approaches, we take a different domain in that we focus on local delay. Actually, the local delay can be interpreted in the same context as the network delay, and can be formalized and calculated based on the studies for the periodic resource scheduling [8], [15]-[17]. But we think that periodic treatments of both VBR and CBR streams needlessly cause longer local delay than non- periodic event-based treatments do, especially when the local system deals with reasonable loads. In the situation when the system suffers just reasonable loads, the non-periodic tasks from a VBR stream can be served quickly enough with a minimum probability of missing deadlines.

In our experiment, as supposed, it is practically shown that the push model takes

shorter delay than the pull model for both PCM [2] constant bit-rate audio and H.261 [3] variable bit-rate video streams. We show that these experimental data are consistent with mathematical interpretations based on M/G/1-multiple vacation queuing models [7]. The vacation is understood as the period while the processor is scheduled to tasks other than the stream under consideration. In this paper, we use the following notation to explain our experimental result:

- $Var[C]$: variance of service period of the pull model. C means service period.
- $Var[V]$: variance of vacation period for the push model. V means vacation period.
- $E[C]$: mean service period of the pull model.
- $E[V]$: mean vacation period of the push model.

In fact, $Var[C]$ and $Var[V]$ were around 10 ms and 0.1 ms, respectively, and $E[C]$ was 100 ms while $E[V]$ spanned from 5 to 15 ms.

This paper is organized as follows. In Section II, we describe a basic mechanism concerning the stream processing which consists of source, destination, stream and filter objects. In Section III, we introduce the pull and push models for the stream processing. Section IV contains experimental data recorded from ComBiStation [9] which provides hardware facilities for real-time H.261 [3] and PCM [2] encoding and decoding. Section V is devoted to the introduction of the M/G/1 multiple vacation queuing models including gated and limited service systems, compares delays of the pull and push models based on these queuing formulas, and shows that the experimental data is consistent with those queuing formulas. In addition to this consistency, we also show that the pull model can take shorter delay when $Var[V]$ was more than 0.5 ms, given that $E[V]$ was in the range of 5 to 15 ms, and $E[C]$ was 100 ms with a variance, $Var[C]$, of 0.1 ms.

II. STREAM PROCESSING

1. Stream Object

A stream is associated with a particular medium. Examples of media include standard raw or compressed media (audio, video, images, graphics, and text) as well as other media streams including mouse/keyboard, pen, animation, and musical instrument digital interface (MIDI) streams. These streams may originate from a file, a device, a connection, or other streams. A stream object reads data from a source object, performs data type conversion, and delivers data to a destination object. Source and destination object mechanisms provide access to multimedia data in a file, device or connection. Data from a source object can be digitally sampled, synthesized, or event driven. For synchronization purposes, the source object is responsible for marking data or time stamping data with a system clock time value.

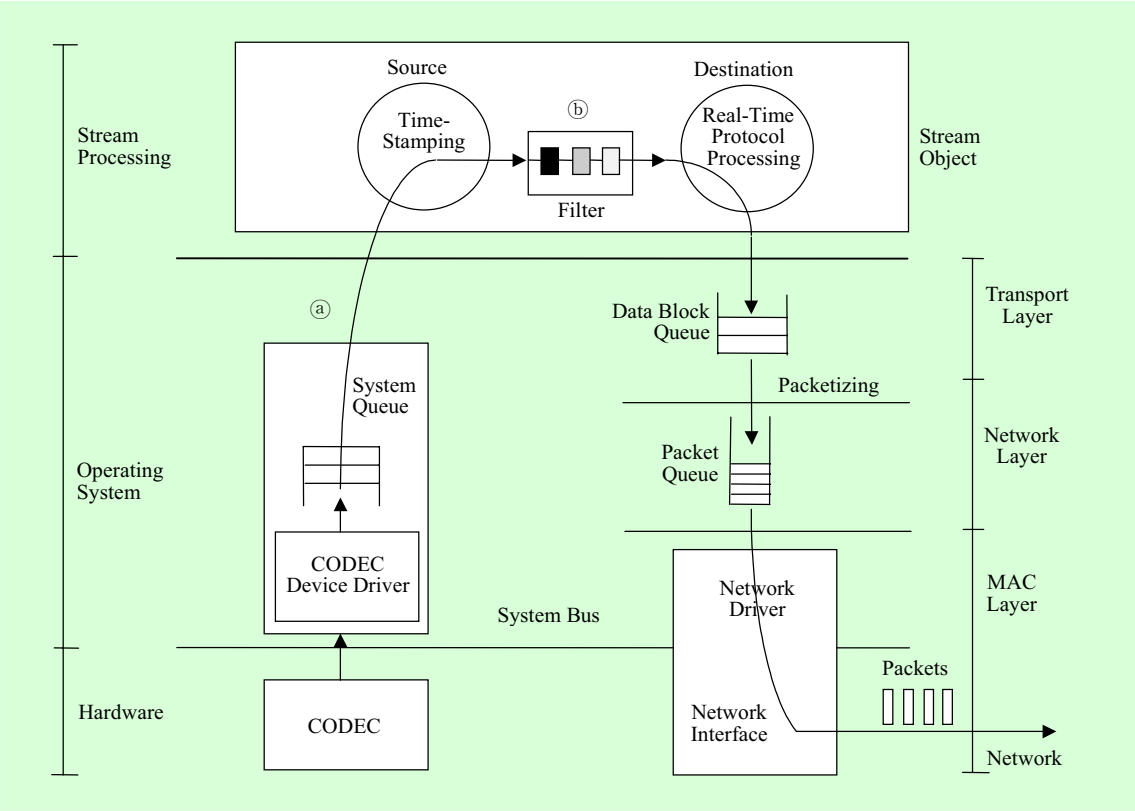

Fig. 8. An example stream flow from a device driver to network.

For streams that originate from a remote site, the time stamp is corrected, within a margin of error, for differences between the remote site and the local site. A level of performance and quality of service between the destination and the source may be specified for each stream. The source and destination objects are different from the sender and receiver in network communication. Actually the sender may have a destination object that provides an access to network connection for sending data, and the receiver may also have a source object to receive data.

2. Filter Objects

Before a stream delivers data to a destination, a filter can perform one of several types of processing operations on it, including format conversion (e.g., RGB images to YUV images), data compression and decompression, and data type conversion (e.g., speech to text). Varying degrees of quality of service and performance can be achieved by having alternate filters for these operations. The basic elements of a filter include an input, an output, control parameters, and a processing program. Filters can be combined to form filter pipes,

or collections of filters. If a filter does not have any control parameters, or if the control parameters are provided at the time of processing such as the quantization table for JPEG compression, then it is said to be context free. A *context-dependent* filter operates within a context that can be specified and controlled independently from the data stream.

3. An Example Stream

Figure 1 illustrates an example stream flow from a device driver for CODEC to a network through a stream object. A frame generated by a compression algorithm in the CODEC hardware is transferred to the device driver, which puts the frame immediately into the system queue. The frame in the queue is to be fetched (ⓐ) based on first-come-first-service discipline by the source object. The source object follows either the pull or the push model which will be explained in the next section. The source object attaches a time-stamp to the fetched frame and transfers it to the destination object through the filter objects (ⓑ) which are supposed to do the process for format conversion, scaling, down-sampling, and so on, for the frame. The destination object finally converts the frame for the real-time protocol which will try to support fast transfer with lower delay and jitter through network media. The real-time protocol and its related topics are out of the scope of this paper.

4. Performance Issues

Among the interconnections of various objects using the primary objects like stream synchronizer, splitter, copier, mixer, binder as well as source, destination and filters, the connection between the device driver and stream object is considered as one of the most sensitive factors for the performance. The performance is primarily concerned with delay and jitter [6]. In this paper, we define the delay as the average waiting time in the system queue. And the jitter is defined as the variation of the delay.

Even though the pull model looks advantageous in terms of scheduling in operating systems since it only offers periodic tasks concerned with a timer, it necessarily suffers delay of half the scheduling period in average. But the push model does not have such intrinsic handicaps. So it is necessary to compare the two models in diverse situations where offered loads, stochastic characteristics, e.g., mean and variances, of the arrival data and their processing time, and so on are subject to changes.

III. IMPLEMENTATION MODELS

We introduce two models in detail, pull model and push model, for implementations concerned with the line ⓐ, in Fig. 1, which is considered as one of the major factors affecting the local delay. The primary difference between these two is that

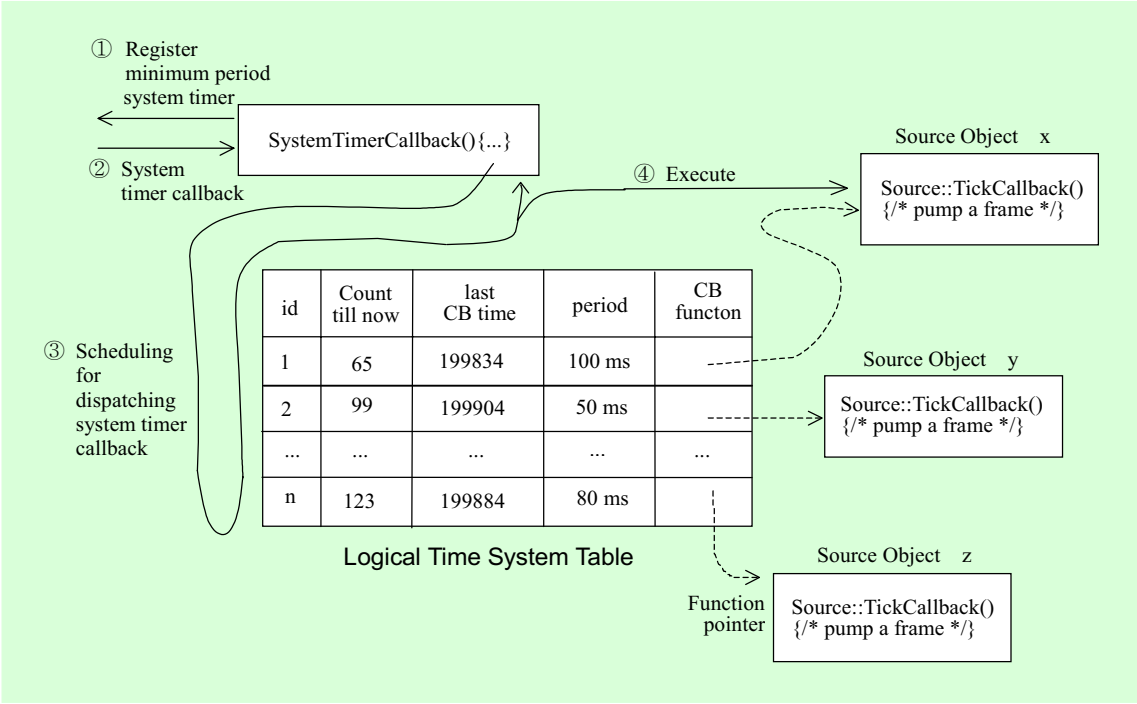

Fig. 9. Logical time system mechanism.

the push model receives events from the device drivers for encoding hardware, while the pull model periodically pumps the incoming data out from the system buffer.

1. Pull Model

The pull implementation model basically uses logical time system (LTS) shown in Fig. 2 as an example situation. LTS deals with system timer events provided by underlying operating system. It dispatches each system timer event to proper source object so that the corresponding stream gets activated periodically. In order to use LTS, the source object registers a callback function Source::TickCallback() to the LTS with tick interval or period called 'source callback interval(SCI).' The LTS manages an LTS table to store the following information:

```
struct LTS_info {
  unsigned int source_id,
  unsigned int callback_count_till_now,
  unsigned int last_callback_time, /* millisecond */
  unsigned int callback_period, /* millisecond */
  (void (*)(unsigned int)) pointer_to_callback_function
}
```

The LTS registers a timer event callback function, SystemTimerCallback(unsigned int current_time) to operating system (see the line ① in Fig. 2). For this registration,

the LTS negotiates with operating system to determine the minimum system timer interval. We call this interval as 'LTS Callback Interval(LCI)'. The operating system is expected to give a callback at every LCI. Upon the registration, the operating system starts to send timer event callbacks (②) to LTS by calling the SystemTimerCallback function. Whenever it receives a callback, the LTS searches the LTS table (③) to calculate temporal distance to the periodical deadline of each source object according to the following fragment of C/C++ codes. The source object with the minimum temporal distance is scheduled to be dispatched the current time event (④).

```
SystemTimerCallback(unsigned int current_time){
  unsigned int min_distance=INT_MAX;
  unsigned int min_distance_id=0;
  int distance, i;

  for(i=1; i<=n; i++)
  {
     distance = abs(LTStable[i].last_callback_time
                    + LTStable[i].callback_period
                    − current_time);
      if(distance<min_distance)
      {
         min_distance=distance;
         min_distance_id=i;
       }
  }

  if(min_dist > system_timer_callback_period)
     return;          /* no source object to be called */
  LTStable[min_distance_id].callback_count_till_now++;
  LTStable[min_distance_id].last_callback_time
   = current_time;

  (* LTStable[min_distance_id].pointer_to_callback_function)
   (current_time);

  return;
}
```

For example, if the current_time is 199,925 ms and the LCI is 15 ms, then the first source object in the LTS table in Fig. 2 is selected to be dispatched current timer event since its SCI is 100 ms and hence the temporal distance is only 9 ms which is the minimum distance.

An overall pull mechanism using LTS is illustrated in Fig. 3. Whenever the callback function Source::TickCallback() is called by LTS, it invokes a member function, say Source::GetFrame(), that performs device-specific operations to fetch a block of data stored in the system buffer allocated to device driver (see the line ⑤ in Fig. 3). After getting a frame, the source object executes a member function, say Stream::DeliverFrame(), to pass the frame to the stream object. The Stream::DeliverFrame() passes it through a series of filters that have been registered with the stream objects, and then passes the filtered data on to the destination object via the Destination::DeliverFrame() member function (⑥).

The aforementioned LTS may miss the deadline of a source object, but the missed object can be rescheduled for next timer event if it is not lagged longer than a certain threshold such as the LCI. Consequently, the LTS creates a probability distribution for SCI. The moments for the distribution are important factors in estimating the average delay incurred by pull model. If these moments get larger, then the average delay gets longer as we explain a queuing theoretic interpretation in Section V.

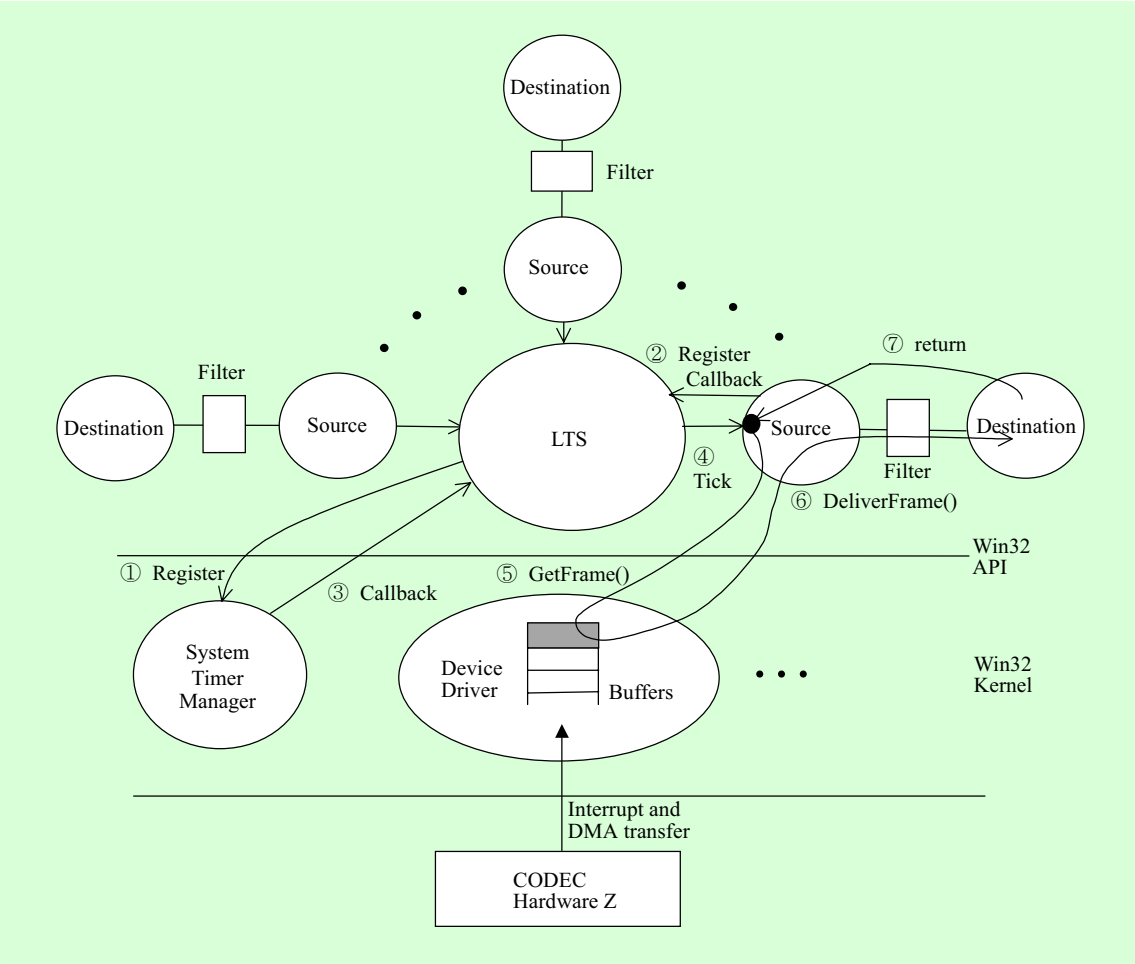

Fig. 10. Pull mechanism.

2. Push Model

While the pull model uses LTS to pump the data periodically at each callback, the push model uses events from device drivers of which scheduling is totally dependent on the underlying operating system. An overall mechanism of push model is illustrated in Fig. 4. The source object registers event callback to corresponding device driver, say device driver Z, and asks that the driver sends an event whenever it finishes a transfer of a data block from the hardware CODEC (see the line ① in Fig. 4) through DMA, and then waits for a new event. When the device driver Z is ready to send an event to the consumer, that is, the source object, it asks the event scheduler in the operating system kernel to deliver an event to the source object (②). The event scheduler, then, schedules events from vari-

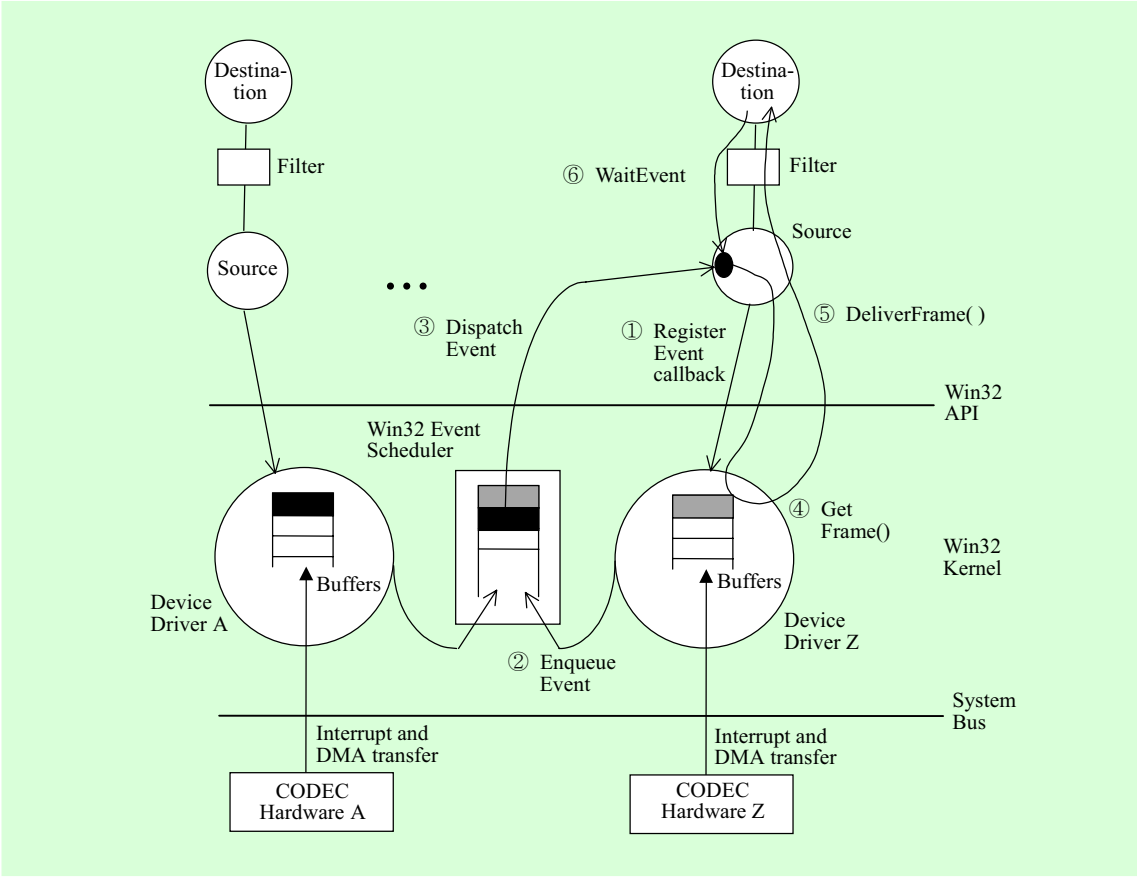

Fig. 11. Push mechanism.

ous event senders according to its own policy, and finally the event from the device driver Z is scheduled to be executed by waking up the source object and delivering the event (③). When the source object receives an event, it performs Source::GetFrame() to read a frame or data block stored at buffers in the device driver Z (④), and delivers the frame to the destination object by passing through filters (⑤). And then it waits next event again (⑥).

The delay of push model is closely related to the event scheduling policy of the underlying operating system, the priority of the device driver process, and current system load, since the delay is incurred by the number of higher priority events waiting to be scheduled by the event scheduler. Therefore, we can intuitively forecast that the push model may be relatively disadvantageous in terms of the delay, when the system suffers heavy load and event traffic. It is shown that this forecast is consistent with the queuing theoretic interpretation in the Section V.

IV. EXPERIMENTAL DELAY ANALYSIS

For our experiment, we defined a common time structure used for both the device driver and the stream object. When the device driver puts frames into the system queue, it writes the system time onto the time structure and attaches it to each frame. Then the stream object fetches a frame according to the pull or push model, and compares the attached time-stamp with the current system time to get the time delay of the frame as a sample data. All the results shown in this paper are based on Windows NT 3.51 operating system, Pentium 100 MHz processors, and EISA bus. We also used a CODEC board [9] developed based on the IIT codec chip set and micro codes for G.711 [2] and H.261 [3].

1. Constant Bit-rate Audio Stream

Figure 5 shows the delay trajectory for the G.711 coded PCM audio stream. Since PCM is a constant bit-rate compression algorithm, the compression hardware was set to send a block of data at every 100 ms. The horizontal and vertical axis represent frame sequences and delay in milliseconds, respectively. Figure 5(a) is for the pull model and shows that the average delay is 364.07 ms and the delay jitter is in the range of about 50 ms. The jitter is due to the clock drifting and other factors depending on the scheduling of the Windows NT 3.51 operating system.

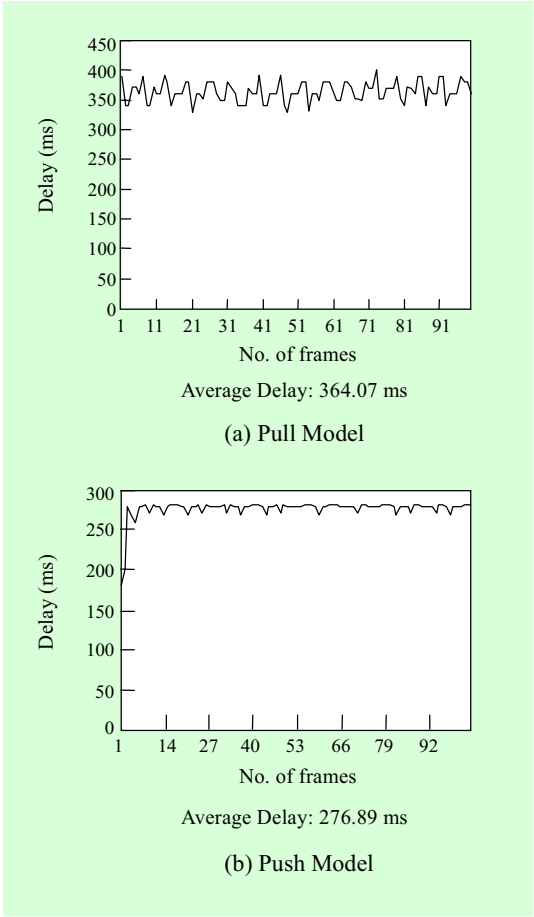

Fig. 12. Average delay for a G.711 constant bit-rate audio stream.

Figure 5(b) is for the push model and shows that the average delay is 276.89 ms and the delay jitter is upper bounded by about 10 ms. This means that the push model outperforms the pull model for the PCM constant bit-rate audio stream in terms of both delay and jitter. Let $D = L_d - S_d$ where L_d is a pull model delay and S_d is a push model delay. Then we can also think that $D = P_l - E_d$, where E_d stands for an event processing delay and P_l stands for

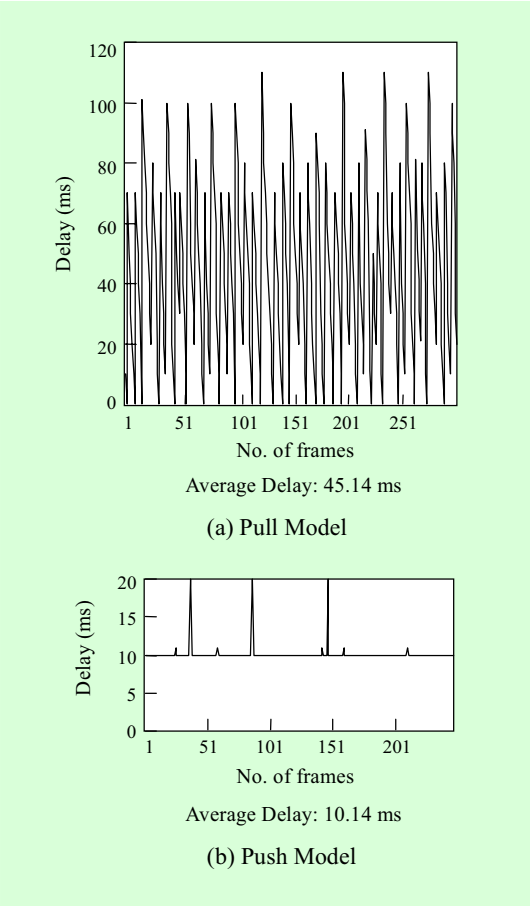

Fig. 13. Average delay for an H.261 constant bit-rate video stream.

the LTS period of the pull model, because the P_l is applied to both hardware and LTS periods. So, in the above case where P_l was set to 100 ms, we can derive $E_d = P_l - (L_d - S_d) = 100 - (364.07 - 27.89) = 12.82$ ms as the event processing delay.

2. Variable Bit-rate Video Stream

Figure 6 shows the delay trajectory for the H.261 video stream. Since H.261 is a variable bit-rate compression algorithm, the compression hardware was set to send a block of data whenever it is ready to transfer to the system queue through the system bus. The horizontal and vertical axes represent frame sequences and delay in milliseconds, respectively. Figure 6(a) is for the pull model and shows that the average delay is 45.14 ms and the delay jitter is in the range of about 100 ms. The jitter is due to the scheduling policies of the Windows NT 3.51 operating system. Figure 6(b) is for the push model and shows that the average delay is 10.14 ms and the delay jitter is upper bounded by about 10 ms. This means that the push model outperforms the pull model for the H.261 variable bit-rate video stream, too, in terms of both delay and jitter.

V. QUEUING THEORETIC INTERPRETATION

The M/G/1-multiple vacation queuing models [7] can be applied to interpret the above two different implementation models and experimental results. The pull model is of gated service systems type, and the push model is of pure limited service systems type. We introduce in this chapter their concepts and formulas on waiting time. In this paper, the vacation is understood as the period while the processor is scheduled to tasks other than the stream under consideration.

1. Gated Service Systems (Pull Model)

In the gated service system, when the server returns from a vacation, it accepts and serves continuously only those messages that are waiting at that time, deferring the service of all messages that arrive during the service period until after the next vacation. In the multiple vacation model, if the server returns from a vacation to find no messages waiting, it begins another vacation immediately, and continues in this manner until it finds at least one message waiting upon returning from a vacation. The length of each vacation is assumed to be independent and identically distributed. Let $Q(z)$ be the probability generating function (PGF) of L, the number of messages found at the end of each vacation, $Q_{(1)}(1)$ be the first moment of the $Q(z)$, and $B^*(s)$ be the Laplace-Stieltjes transform (LST) of the distribution function (DF) of the service time, $B(x)$. Then the LST of the waiting time distribution is as follows [7], assuming that the service discipline is first-come-first-served (FCFS):

$$W^*_{FCFS}(s) = \frac{\lambda\{Q[B^*(s)] - Q(1-s/\lambda)\}}{Q^{(1)}(1)[s-\lambda+\lambda B^*(s)]} . \quad (19)$$

Because of gated service, L consists of the number h of messages in the system when the server leaves and the number f of messages that arrive during a vacation:

$$L = h + f \ . \quad (20)$$

We assume that h and f are independent. If $H(z)$ and $F(z)$ denote the PGF's of h and f, respectively, this independence means that

$$Q(z) = H(z)F(z) \ . \quad (21)$$

Let $V^*(s)$ be the LST of the DF for the vacation time V, and let $S^*(s)$ be the LST of the DF for the length S of a service period. From the mechanism of gated service, we get

$$S^*(s) = Q[B^*(s)] \ . \quad (22)$$

The PGF of the number h of messages in the system when a vacation starts in given by

$$H(z) = S^*(\lambda - \lambda z) \ , \quad (23)$$

and the PGF of the number f of messages that arrive during the vacation is given by

$$F(z) = V^*(\lambda - \lambda z) \ . \quad (24)$$

Assuming that these two numbers h and f are independent, we substitute (4)$\sim$(6) into (3) to obtain the functional relationship

$$Q(z) = Q[B^*(\lambda - \lambda z)]V^*(\lambda - \lambda z) \ . \quad (25)$$

Let $C^*(s)$ be the LST of the DF for the service cycle time C, which is defined as the time interval between the terminal points of two successive vacations. (A service cycle consists of a service period and a vacation that follows the service period.) From the definition of the gated service, we have the relationship in FCFS service discipline

$$Q(z) = C^*(\lambda - \lambda z) \ , \quad (26)$$

which implies that the number of messages present in the system at the end of a vacation equals the number of messages that arrive during the preceding service cycle. Equation (7) is then rewritten as

$$C^*(s) = C^*[\lambda - \lambda B^*(s)]V^*(s) \ . \quad (27)$$

The mean cycle time is given by

$$E[C] = \frac{E[L]}{\lambda} = \frac{E[V]}{1-\rho} = E[S] + E[V] \ . \quad (28)$$

Therefore, (1) can be rewritten in terms of the distribution of C as

$$W_G^*(s) = \frac{C^*[\lambda - \lambda B^*(s)] - C^*(s)}{E[C][s - \lambda + \lambda B^*(s)]} \ , \quad (29)$$

and finally, from (11) we can get

$$E[W_G] \stackrel{\triangle}{=} \left. \frac{dW_G^*(s)}{ds} \right|_{s=0} = \frac{(1+\rho)E[C^2]}{2E[C]}$$
$$= \left(\frac{1+\rho}{2}\right)\left(\frac{Var[C]}{E[C]} + E[C]\right). \quad (12)$$

2. Limited Service Systems (Push Model)

In limited service systems, the number of messages that are served continuously during a service period is limited. In the (pure) limited service system with multiple vacations, the server takes a vacation each time it completes service to a single message. If there are no messages waiting in the system when the server returns from a vacation, it takes another vacation. Vacations are repeated until at least one message is found at the end of the vacation. Let $V^*(s)$ be the LST of the length of a vacation V. The LST $W_K^*(s)$ of the DF for the message waiting time in the FCFS limited service system is obtained by replacing $B^*(s)$ with $B^*(s)V^*(s)$ in the formula for the FCFS exhaustive service system with multiple vacations [7] as

$$W_K^*(s) = \frac{1 - V^*(s)}{sE[V]} \cdot \frac{s(1-\rho-\lambda E[V])}{s - \lambda + \lambda B^*(s)V^*(s)} \ , \quad (13)$$

and the mean message waiting time is given by

$$E[W_K] \stackrel{\triangle}{=} \left. \frac{dW_K^*(s)}{ds} \right|_{s=0} = \frac{1}{2}\left(\frac{Var[V]}{E[V]} + E[V]\right)$$
$$+ \frac{\lambda}{2}\left(\frac{b^{(2)} + 2\lambda E[V] + Var[V] + E[V]^2}{1 - \rho - \lambda E[V]}\right) \quad (14)$$

where $b^{(i)}$ is the ith moment of the service time distribution, $B(x)$.

By comparing (12) and (14), we can predict performances of each implementation approach as well as interpret the experimental data. As common features of the two equations, we see that the waiting time of both models, $E[W_G]$ and $E[W_K]$, monotonically increase when the ρ increases. $E[W_G]$ linearly increases with the increment of $Var[C]$, and so does $E[W_K]$ with $Var[V]$.

Figure 7 shows that the pull model suffers longer waiting time than the push model when the average vacation period of the push model, $E[V]$, does not exceed a threshold which can be derived from (12) and (14). If the period, $E[V]$, exceeds the threshold, then the pull model obtains a shorter waiting time than the push model. The threshold increases when the variance of the service cycle time of the pull model, $Var[C]$, gets larger. This means that the

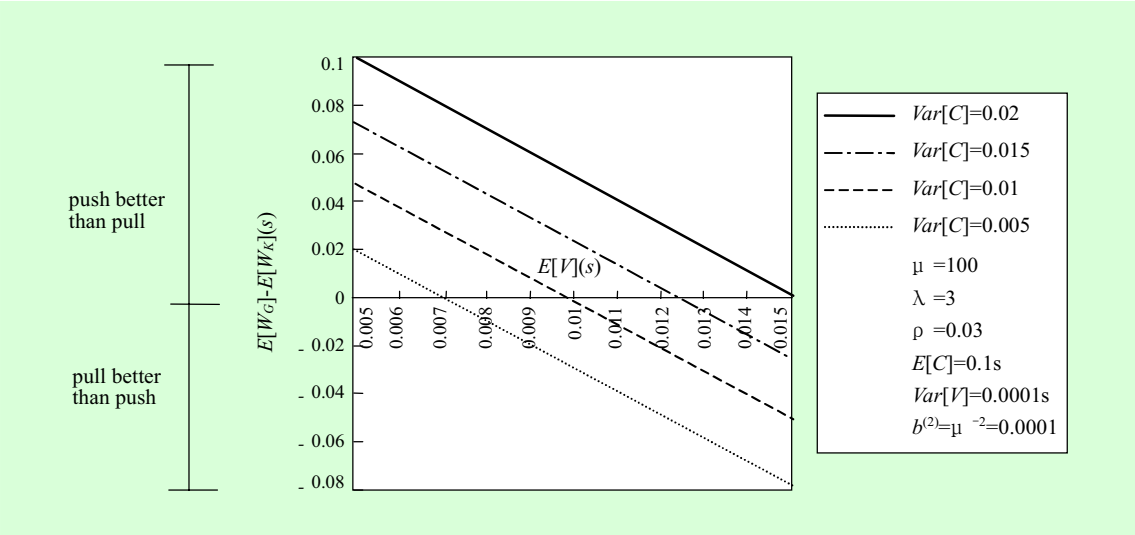

Fig. 14. Waiting time differences with variances of service cycle time C.

push model is beneficial when the system clock that the pull model uses suffers irregularity (large $Var[C]$). This usually happens in commercial non-real-time operating systems where their schedulers do not support the earliest-deadline-first policies.

Figure 8 shows that the push model suffers longer waiting time than the pull model when the variances of the vacation period of the push model, $Var[V]$, get larger. If the variance, $Var[V]$, exceeds a certain level, then the push model never outperforms the pull model. This usually happens when the system suffers irregular bulk traffics or tasks. The pull model obtains a shorter delay than the push model when $Var[V]$ is more than 0.5 ms that is about 10 percent of $E[V]$, given that $E[V]$ is in the range from 5 to 15 ms, and $E[C]$ is 100 ms with the variance, $Var[C]$, 0.1 ms. In contrast, the push model obtains a shorter delay when $Var[C]$ is more than 20 ms, given that $E[V]$ is in the same range, $E[C]$ is 100 ms, and $Var[V]$ is 0.1 ms.

The experimental data shown in Figs. 5 and 6 show that the average delay of the pull model is longer than that of the push model. On the basis of the above discussion with the M/G/1-vacation queuing models, this can be interpreted that the system used is not suffering any irregular bulky tasks and the arrival rates of audio and video data are small enough. Actually, the $Var[C]$ is more than 10 ms while $Var[V]$ is minimum, and the $E[C]$ is set to 100 ms, in our implementation. All the compression and decompressions are done in specialized hardware so that the main processor is left free from such computation-oriented tasks, hence consequently deals only with a small

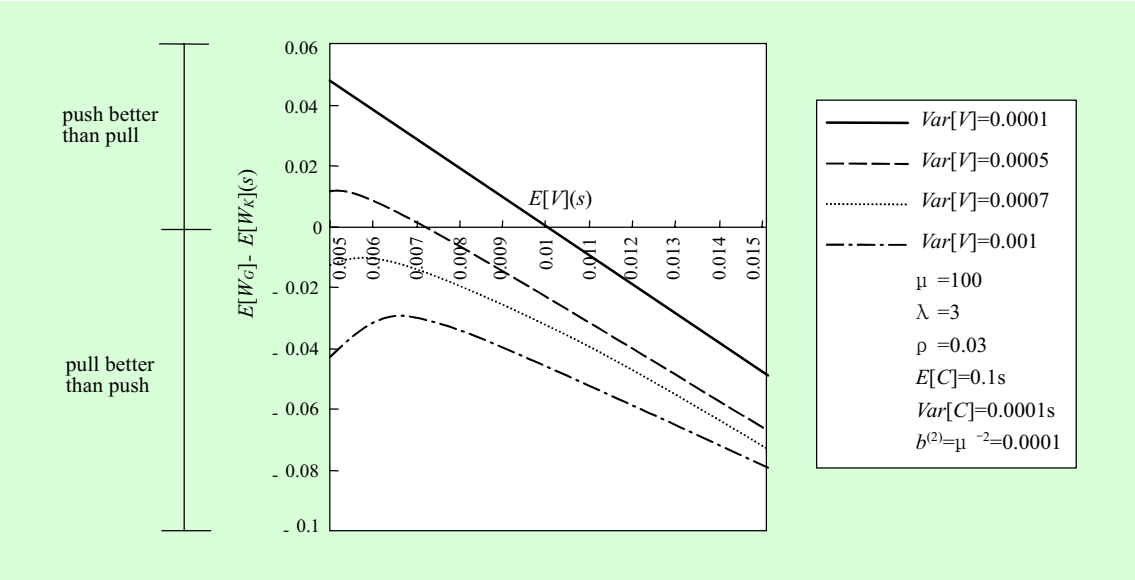

Fig. 15. Waiting time differences with variances of vacation period V.

offered load, ρ, lower than 0.03. These facts make the experimental data consistent with the above queuing theoretical interpretations.

VI. CONCLUSIONS

In this paper, we investigated two implementation approaches, pull model and push model, for handling continuous input streams from real-time compressors, G.711 and H.261. While the pull model periodically pumps out the incoming data from the system buffer, the push model receives events from the device drivers for encoding hardware. We experimentally compared the performance of each in terms of the delay. We conclude that the push model outperforms the pull model in usual situations for both PCM constant bit-rate audio and H.261 variable bit-rate video streams. This result was supported by the M/G/1-vacation queuing theories in this paper, too. And we also see that the push model may obtain a longer delay than the pull model when the system suffers bulky tasks so that the variance of vacation periods for the push model exceed a certain threshold.

In the future, we plan to apply this result for efficient usage of transport protocols [20]-[23]. In the view of the transport layer, the push model forms M/G/1 queue, while the pull model forms D^X/G/1 queue where the interarrival time is deterministic but each arrival forms a batch of frames. The number of frames in a batch may follow the gamma distribution [24]. We expect that the push model outperforms the

alternative in normal transport situations so that the end-to-end delay is reduced, too.

REFERENCES

[1] R. Baker, A. Downing, K. Finn, E. Rennison, D-H. Kim, and Y-H Lim, "Multimedia processing model for a distributed multimedia I/O system," *Proc. of 3rd Int'l Workshop on Networking and Operating Systems for Digital Audio-Video*, San Diego, California, 1993.

[2] ITU-T Recommendation G.711, *Pulse Code Modulation (PCM) of Voice Frequencies*, ITU, 1988.

[3] ITU-T Recommendation H.261, *Video CODEC for Audiovisual Services at px64 kbits/s*, 1995.

[4] D-H. Kim, S-H. Ohe, J-K. Hwang, Y-H. Lim, and E. Rennison, "A synchronization and integration model for audio, video, and time-based graphics multimedia," *Proc. of the 2nd Pacific Rim Conference on Artificial Intelligence*, vol. II, Seoul, Korea, 1992, pp. 1093-1099.

[5] D.-H. Kim, *MuX User's Manual*, MuX User's Group, 1995, *http://mux.etri.re.kr* (current Sep. 26, 1997).

[6] D. L. Stone, and K. Jeffay, "Queue monitoring: a delay jitter management policy," *ACM J. of Multimedia*, pp. 151-162, 1994.

[7] H. Takagi, *queuing Analysis: A Foundation of Performance Evaluation, vol. 1: Vacation and Priority Systems*, Part 1, Amsterdam: North-Holland, 1991.

[8] W. Zakowski, *Reference Architecture Model Specification*, Approved Rev. 1.0, MMCF/95-009, Multimedia Communication Forum, 1995.

[9] Y-H. Lim, "ComBiStation: a computer platform for a distributed multimedia computing environment," *J. (C) of Korea Information Science Society*, vol. 2, no. 2, pp. 160-181, 1996.

[10] P. Forsyth, *MMCF Middleware*, MMCF/95-005 Draft Rev. 4.0, Multimedia Communication Forum, June 1996.

[11] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *J. ACM*, vol. 20, pp. 47-61, 1973.

[12] C. Vogt, "Quality-of-service management for multimedia streams with fixed arrival periods and variable frame sizes," *ACM J. of Multimedia*, pp. 66-75, 1995.

[13] D. Ferrari, A. Banerjea, and H. Zhang, *Network Support for Multimedia: A Discussion of the Telnet Approach*, Technical Report 92-072, International Computer Science Institute, Berkeley, CA., 1995.

[14] M. Moran, and D. Wolfinger, *Design of a Continuous Media Data Transport Service and Protocol*, Technical Report 92-019, International Computer Science Institute, Berkeley, CA, 1992.

[15] R. Steinmetz, "Analyzing multimedia operating system," *IEEE Multimedia Magazine*, Spring 1995.

[16] R. G. Herrtwich, "The role of performance, scheduling, and resource reservation in multimedia systems," *Operating Systems of the 90s and Beyond*, A. Karshmer and J. Nehmer, Eds., Lecture Notes in Computer Science, Springer-Verlag, 1991, pp. 279-284.

[17] C. W. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves: operating system support for multimedia applications," *Proc. of the Int'l Conf. on Multimedia Computing and Systems*, Boston, MA, 1994, pp. 90-99.

[18] *Multimedia Communication Quality of Service*, Working Document ARCH/QOS/94-001 Rev. 2.0, Multimedia Communication Forum, March 4, 1995.

[19] A. Campell, G. Coulson, and D. Hutchinson, "A quality of service architecture," *Proc. of ACM SIGCOMM*, April 1994, pp. 6-27.

[20] B. Sabata, M. Brown, and Barbara A. Denny, "Transport protocol for reliable multicast: TRM," *Proc. of Int'l Conf. on Networks*, Orlando, Florida, Jan. 8-10, 1996.

[21] T. Mongomery, *Design, Implementation, and Verification of the Reliable Multicast Protocol*, master's thesis, West Virginia University, Department of Statistics and Computer Science, 1994.

[22] J. Pasquale, G. C. Polyzos, and G. Xylomenos, "The multimedia multicast problem," accepted for publication, *ACM J. of Multimedia System*, 1997.

[23] ITU-T Recommendation H.323, *Visual Telephone Systems and Equipment for Local Area Networks Which Provide a Non-Guaranteed Quality of Service*, ITU, 1996.

[24] D. Heyman, Ali Tabatabai, and T. V. Lakshman, "Statistical analysis and simulation study of video teleconference traffic in ATM networks," *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 2., no. 1., pp. 49-59, 1992.

[25] *Media Device Interface*, Working Document MMCF/96-013 Rev. 2.0, Multimedia Communication Forum, Oct. 1996.

Doo-Hyun Kim received his B.S. degree from Seoul National University (1985) and M.S. degree from Korea Advanced Institute of Science and Technology (1987). He is now a Principal Member of Research Staff of Electronics and Telecommunications Research Institute (ETRI), and also Ph. D. student of Computer Science Department of KAIST. He was an International Fellow at SRI International from 1991 to 1993. Mr. Kim's research interests include performance analysis of distributed multimedia system, multimedia middleware, multimedia operating system, distributed multimedia hyperpresentation, quality of service management, and real-time multimedia support for worldwide webs.

Kyung Hee Lee received his B.S. and M.S. degrees from Kyungpook National University (1992). He is now a member of research staff of Electronics and Telecommunications Research Institute (ETRI), and is working on developing an Internet audio video teleconferencing system based on ITU-T H.323 and T.130. His research interests include 3D audio, multimedia middleware, and real-time multimedia system.

Sang Hwan Kung received the B.S. degree in computer science from Soongsil University in 1977 and the M.S. degree in management information system from Korea University in 1983. He joined the Military Automatic Data Processing Center in 1977 and worked for computerization of army logistic system for four years. He joined ETRI in 1982 and conducted research and development in the areas of telecommunication network management, minicomputer called TICOM, and multimedia middleware and applications like desktop conferencing and video mail systems. He is currently interested in the distributed multimedia systems.

Jin Hyung Kim received his B.S. degree from Seoul National University, and his M.S. and Ph.D. degrees from the University of California, Los Angeles (1983). He is now the President of Korea R&D Information Center (KORDIC), and also Prof. of Computer Science Department of Korea Advanced Institute of Science and Technology (KAIST). He worked with Hughes AI Center, Calabasas, as a Senior Staff Member from 1981 to 1985, and was a visiting scientist at IBM T. J. Watson Research Center.