An Integrated Push/Pull Buffer Management Method in Multimedia Communication Environments

Sungyoung Lee[†], Hyon Woo Seung[††], Tae Woong Jeon[†††]

[†] *Department of Computer Engineering, KyungHee University, Seoul, Korea*
{*sylee@oslab.kyunghee.ac.kr*}
[††] *Department of Computer Science, Seoul Women's University, Seoul, Korea*
{*hwseung@swu.ac.kr*}
[†††] *Department of Computer Science, Korea University, Seoul, Korea*
{*jeon@kusccgx.korea.ac.kr*}

Abstract

Multimedia communication systems require not only high-performance computer hardware and high-speed networks, but also a buffer management mechanism to process voluminous data efficiently. Two buffer handling methods, push and pull, are commonly used. In the push method, a server controls the flow of data to a client, while in the pull method, a client controls the flow of data from a server. These two buffering schemes can be applied to the data transfer between the packet receiving buffer, which receives media data from a network server, and media playback devices, which play the received media data. However, the buffer management mechanisms at client-side mainly support only one of the push and the pull methods. In other words, different types of playback devices separately use either but not both of the buffer methods. This leads to inefficient buffer memory usage and to inflexible buffer management for the various types of media playback devices.

To resolve these problems, in this paper we propose an integrated push/pull buffer mechanism able to manage both push and pull schemes in a single buffer at client-side. The proposed scheme can support various media playback devices using a single buffer space, which in consequence saves memory space compared to the case where a client keeps two types of buffers. Moreover, it facilitates the single buffer as a mechanism for absorbing network jitter effectively and efficiently. The proposed scheme has been implemented in an existing multimedia communication system called ISSA developed by the authors, and has showed good performance compared to the conventional buffering methods in multimedia communication environments.

1. Introduction

A flexible buffer management mechanism is required to process buffer I/O's efficiently, to calculate an adequate buffer size according to the application service and the media type, and to control the buffer over/underflow effectively.

Two buffer handling methods, Server-push (or just push) and Client-pull (or just pull), are commonly used, depending on which side has control over the data flow. In the push method, a server controls the flow of data and periodically transfers appropriate data to clients. Although the push method is suitable for broadcast services, the server must control the data transmission speed (bit-rate) in order to avoid buffer over/underflow at the clients' side. The pull method is a kind of polling method where a client, having control over the data flow, requests data to a server and the server transfers the requested data to the client. Although the client can control buffer over/underflow, the pull method is suitable only for unicast services, not for broadcast services [1,2,3].

The two methods can be applied to data transfer not only between network server and clients, but also between the packet receiving buffer, which receives media data from the network server, and media playback devices, which play the received media data. However, most buffer management mechanisms at the clients' side are usually focused on network considerations such as the buffer size control according to end-to-end network situations, and mainly support only one of the push and the pull methods. Consequently, they have some limitations in supporting various media playback devices. Even though some of them support both methods, it is difficult to utilize a variety of devices since they do not provide a unified structure. In an integrated multimedia communication system such as ISSA (Integrated Streaming Services Architecture)[4], various media playback devices cannot be supported if the packet receiving buffer provide either of the push or pull modes. A flexible buffering mechanism is also needed to absorb end-to-end network jitter, in a network environment like Internet with frequent packet loss and unstable bandwidth.

In this paper, we propose an efficient and flexible push/pull buffer management mechanism for the client-side, which readily supports various media playback devices by providing a unified interface for both push and pull modes at the client's packet receiving buffer, and easily absorbs end-to-end network jitter. Using the proposed scheme, can obviate the need to install an extra stub for buffer passing appropriate to each media device; furthermore, better efficiency in memory use can be accomplished by also letting the buffering function absorb network jitter, instead of buffering only.

This paper is organized as follows. After briefly reviewing related work in Section 2, we sketch the design architecture of the proposed buffer management scheme in Section 3. The implementation and experimental results of the scheme are described in Section 4. We present our conclusions in Section 5.

2. Related Work

Although much work has been done in the area of client-side buffer management techniques, not much research has been reported on push/pull mode buffer management related to data passing to media playback devices.

The JMF (Java Media Framework) of Sun Microsystems[5] offers push/pull buffer management methods through two interfaces, pushDataSource and pullDataSource. It is, however, not easy to program

multimedia applications using JMF since it is not furnished with a unified interface for push/pull methods.

Acharya et al. at Brown University have studied ways to efficiently combine the push and pull approaches[6]. This method is focused upon supporting both push and pull methods among network server and clients, where the pull method is provided simply to make up for the weak points of the push method. In that sense, it is different from the scheme proposed in this paper which provides a unified structure to support both methods on the client-side.

The buffer management scheme proposed in DAVIC (Digital Audio VIsual Council)[9] is quite similar to the one proposed at Brown University[6]. However, the clients in the scheme not only request data through the back-channel, but also notify the server of the buffer states and gather the data needed in advance.

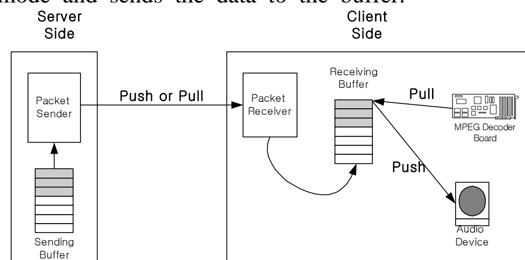
Jack Lee at Hong Kong University has proposed an efficient buffer management scheme in which the optimal buffer size is calculated to prevent server's buffer underflow and clients' buffer overflow, in his study of a concurrent push scheduling algorithm for push-based parallel video servers[7]. Even though this scheme shows a good performance by utilizing a parallel transmission method, it requires higher buffer capacity. Again, these two methods are different from the one we propose.

3. Design of the Proposed Scheme

In this section, we explain the operational architecture of the proposed scheme and the structure of the internal buffers, and summarize the buffer management algorithms.

3.1 The Design Model

Figure 1 illustrates the operational architecture of the push/pull buffer management scheme. It shows how the client-side push/pull scheme works with various kinds of media playback devices. The packets transmitted by the Packet Sender from the server through push or pull method are received and translated by the Packet Receiver of the client. Then, the media stream data are sent to the Receiving Buffer, and reproduced by the media playback devices through either the push or pull method initially set. To each of the media playback devices, an appropriate data transmission method is initially assigned. When the transmission session begins, the data stream is transmitted to the corresponding device, according to the method previously set. The Receiving Buffer on the client-side works regardless of the network media transmission methods, since the Packet Receiver receives data from the server through push or pull mode and sends the data to the buffer.



[Figure 1] Operational Architecture of Proposed Scheme

The proposed scheme supports both push and pull mode in a single buffer.

3.2 Algorithms

The proposed buffer management scheme consists of 7 algorithms; two for buffer initialization and removal, another two buffering algorithms to absorb network jitter, and three algorithms for the data I/O.

Figure 2 shows the pseudo-code for the *InitBuffer* algorithm. It must first be determined which buffer passing mode (push or pull) the media device supports. Then, an appropriate buffer size is calculated using the following formula:

*buffer size = bitrate of given media * buffering time * scale factor*

where *bitrate of given media* is the playback speed of the given media (bit/secs), *buffering time* is the buffering time in seconds for the media, and *scale factor* is used to calculate an actual buffer size. The scale factor is used to get a "safe" buffer size to prevent overflow of the receiving buffer, and is usually greater than 1.0. If the factor is 1.0, the actual buffer size is equal to the ideal buffer size. After the buffer size is calculated, memory allocation is executed for the buffer. Then, the *StartBuffering* algorithm is performed.

InitBuffer Algorithm
Determine buffer passing mode for media device to use
**) buffer passing mode = push or pull*
Calculate buffer size
**) buffer size = bitrate of the given media * buffering time * scale factor*
Allocate memory for buffer
Invoke StartBuffering algorithm

[Figure 2] InitBuffer Algorithm

The *DeinitBuffer* algorithm, shown in Figure 3, is used to stop media playback and remove the buffer. If buffering is in process (buffering mode is *start*), the *StopBuffering* algorithm is invoked. Then, waiting until all media stream data in buffer are consumed, it frees memory for the buffer.

DeinitBuffer Algorithm
If buffering mode == start then
Invoke StopBuffering algorithm
Wait until all media stream data in buffer are consumed
Free memory for buffer

[Figure 3] DeinitBuffer Algorithm

The *StartBuffering* algorithm begins by changing the buffering mode to *start*. Then, the current time is saved to calculate the playback delay time. The push/pull operations are locked until the total size of data in the buffer exceeds the buffering size. Then, *StopBuffering* is invoked[Figure 4]. The buffering size is calculated using the following formula: *buffering size = bitrate of the given media * buffering time*

StartBuffering Algorithm
Change buffering mode to start
Save current time for calculation of playback delay time
Lock push/pull operations until buffer data size exceed buffering size
**) buffer data size = total size of data in buffer*
**) buffering size = bitrate of the given media* buffering time*
Invoke StopBuffering algorithm

[Figure 4] StartBuffering Algorithm

The *StopBuffering* algorithm, shown in Figure 5, is invoked at the end of *StartBuffering*.

Stop Buffering Algorithm
Update playback delay time by adding additional playback delay time
**) additional playback delay time = current time-saved buffering start time*
**) playback delay time = playback delay time + additional playback delay time*
Unlock push/pull operations
Change buffering mode to stop

[Figure 5] StopBuffering Algorithm

First, it calculates the additional playback delay time by subtracting the saved buffering start time from the current time, and updates the playback delay time by adding the additional playback delay time. Next, it unlocks push/pull operations by setting up a timer event for the next push mode operation if the current buffer passing mode is *push*, or by waking up the sleeping pull mode operation if the mode is *pull*. Finally, it changes the buffering mode to *stop*.

The *AddData* algorithm is shown in Figure 6. First, it checks whether buffer overflow has occurred. If so, it drops new incoming data and terminates the execution. If not, it copies new incoming data into the buffer. Then, if the buffer passing mode is *push*, it appends the info block corresponding to the new incoming data to the info block linked list.

```
AddData Algorithm
Lock buffer
Apply playback delay time to calculate expire time of
    new incoming data
If (size of new incoming data + buffer data size >
        buffer size) then
        Buffer overflow occurred
        Drop new incoming data
        Unlock buffer
        Terminate this algorithm
Copy new incoming data into Buffer
Adjust pointer to stream tail with position of new data
If (buffer passing mode == push) then Add an info
    block corresponding to new incoming data.
Update buffered data size
Unlock buffer
```

[Figure 6] AddData Algorithm

Note: In terms of the QoS (Quality of Service), it is not desirable for the algorithm to drop any incoming data in case buffer overflow occurs. To resolve this problem, we could introduce of a scheme that saves the surplus data in a temporary buffer and moves them to the real buffer later. However, due to the overhead to manage the temporary buffer, real-time processing of the media data may not be guaranteed.

Figure 7 shows the *pushData* algorithm which sends the data in buffer to the media playback device in push mode.

```
pushData Algorithm
Lock buffer
If (buffer data size < size of data to push) then
        Buffer underflow occurred
        Invoke StartBuffering algorithm
        Terminate this algorithm
Push data of stream head in buffer to media device
Remove info block corresponding to removed data
Change pointer to stream head in buffer
Set up next timer event for pushData algorithm
Unlock buffer
```

[Figure 7] pushData Algorithm

First, it checks whether buffer underflow has occurred. If so, it restarts buffering and terminates the execution. If not, it pushes the data to the media device and removes the data from the buffer by changing the pointer to stream head. Next, it sets up a timer event for the next *pushData* operation.

Figure 8 shows the *pullData* algorithm. First, it checks whether buffer underflow has occurred. If so, it restarts buffering and waits until buffering stops. If not, it pulls the data to the media device and removes the data from the buffer by changing the pointer. Unlike *pushData*, *pullData* is invoked from the media device interface without using a timer event.

```
pullData Algorithm
Lock buffer
If (buffer data size < size of data to pull) then
        Buffer underflow occurred
        Invoke StartBuffering algorithm
        Sleep until buffering stop
Pull data of stream head in buffer to media device
Change pointer to stream head in buffer
Unlock buffer
```

[Figure 8] pullData Algorithm

4. Performance Evaluation

In this section, we explain the implementation process and experimental results of the proposed scheme.

4.1 Implementation Environment

Our scheme has been implemented in an existing multimedia communication system, called ISSA [4] in which various streaming applications such as VOD and AOD can be easily produced. ISSA provides many functions concerned with content management, transmission protocols and media processing for various streaming application programs. Among many components in the ISSA, *Media Manager* plays an important role offering various media processing functions such as media file processing (MPEG, WAV, AU, etc.), database-stored media stream processing, A/V Codec for encoding/decoding media data to other formats, and controlling A/V devices. It consists of *Media Source* and *Media Sink*.

Since the *Media Sink* component plays the role of receiving the media stream data from the client, and sending them to the appropriate media playback devices, the proposed buffer management scheme is designed and implemented to be incorporated in *Media Sink*. In order to test the performance of the scheme, we built a simple server program sending media stream data stored in files, and a client program playing the data. Both programs are written in C++. The server program works on MS Windows-NT or Sun Solaris 2.X, and the client program on MS Windows-NT. The data transmission between the server and the client is done using the RTP (Real-Time Transport Protocol)[9] which was made for the real-time media data transmission over a TCP/IP-based 10/100 Mbps Ethernet.

4.2 Experimental Results

In order to evaluate the performance and measure the overhead of the proposed scheme implemented in ISSA, we have performed three experiments. In the first two experiments, real media playback devices were used to test whether the data transmission and buffering in the system works well with both push and pull methods. In the last

experiment, we created virtual software media playback devices to examine the performance overhead of the scheme. Tables 1 and 2 show the attributes of the media data used in the experiments.

[Table 1] Attributes of Test media A used in Experiments

Encoding Type	MPEG-1 System Layer Stream
Bitrate	1,715,200 bps
Resolution	320x240

[Table 2] Attributes of Test media B used in Experiments

Encoding Type	Linear PCM Audio
Bitrate	1,411,200 bps
Sampling Rate	44,100 Hz
Number of Channels	2
Bits Per Sample	16

The network testbed used to test the remote playback function when the server and the client are on separate hosts.

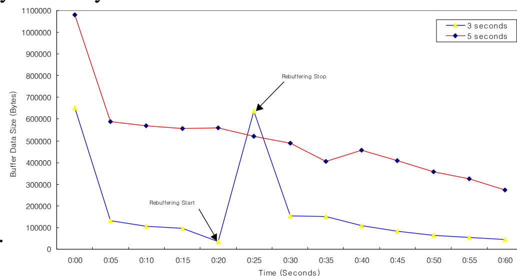
4.2.1 Experiment 1

In the first experiment performed upon the testbed, we tested whether the proposed buffer management scheme operates correctly in pull mode. The scheme in the client-side sends the test media data in the receiving buffer, with the attributes in Table 2, to the DirectShow media playback device in pull mode. Table 3 shows the parameter values for Experiment 1. For each of 3 and 5 second Buffering Time (BT), we examined the operation of the scheme in pull mode with two cases of the Buffer Scale Factor(BSF), 1.1 and 1.3.

[Table 3] Buffer Parameter Values for Experiment 1

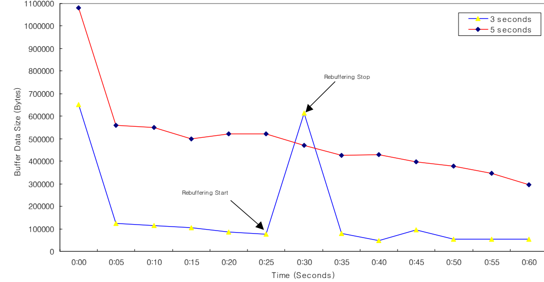
BT	3 seconds		5 seconds	
Buffering Size	1,715,200 bits / 8*3 = 643,200 bytes		1,715,200 bits / 8*5 = 1,072,000 bytes	
BSF	1.1	1.3	1.1	1.3
Buffer Size	643,200* 1.1 = 707,520 bytes	643,200* 1.3 = 836,160 bytes	1,072,000* 1.1 = 1,393,600 bytes	1,072,000* 1.3 = 1,393,600 bytes

As shown in Figures 9 and 10, rebuffering caused by network packet delay occurred once during 1 minute of playing time in the case of 3-second Buffering Time, while no rebuffering occurred and the Buffer Data Size continuously decreased in the case of 5-second Buffering Time. The transmission delay time can be very long for high-bandwidth media streams like the MPEG-1 System Layer stream.



[Figure 9] Experiment 1 (A): Pull, Remote Playback, Buffer Scale Factor (1.1)

On the other hand, changing Buffer Scale Factors had little influence on the operation of the scheme. The reason was that the media processing speed in the client side was so fast that the data in the buffer could have been processed in time without delay



[Figure 10] Experiment 1 (B): Pull, Remote Playback, Buffer Scale Factor (1.3)

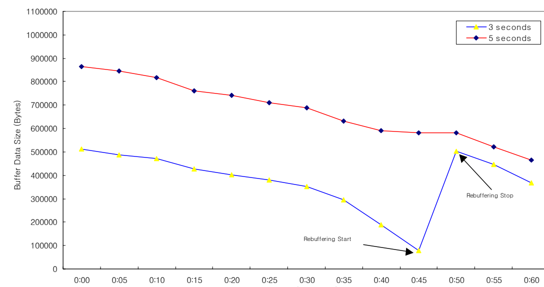
4.2.2 Experiment 2

In the second experiment that was performed upon the testbed as in the first experiment, we tested whether the proposed buffer management scheme operates correctly in push mode. The scheme in the client-side sends the test media data in the receiving buffer, with the attributes in Table 2, to the PCM audio playback device in push mode.

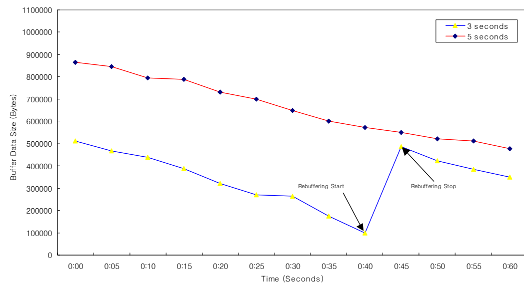
Table 4 shows the parameter values for Experiment 2. As in Experiment 1, for each of 3 and 5 second Buffering Time, we examined the operation of the scheme in push mode with two cases of the Buffer Scale Factor, 1.1 and 1.3. As can be seen in Figures 11 and 12, rebuffering occurred once in the case of 3-second Buffering Time, while no rebuffering occurred in the case of 5-second Buffering Time. From the first and second experiments, it can be said that the proposed scheme can absorb the network jitter regardless of the data passing mode of the buffer (push or pull), providing flexible buffering functions to overcome buffer over/underflow.

[Table 4] Buffer Parameter Values for Experiment 2

BT	3 seconds		5 seconds	
Buffering Size	1,411,200 bits/8*3 = 529,200 bytes		1,411,200 bits/8*5 = 882,000 bytes	
BSF	1.1	1.3	1.1	1.3
Buffer Size	529,200* 1.1 = 582,120 bytes	529,200* 1.3 = 687,960 bytes	882,000* 1.1 = 970,200 bytes	882,000* 1.3 = 1,146,600 bytes



[Figure 11] Experiment 2 (A): Pull, Remote Playback, Buffer Scale Factor (1.1)



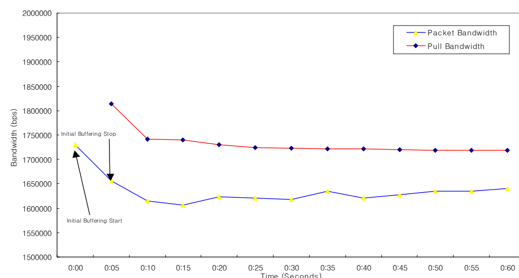
[Figure 12] Experiment 2 (B): Pull, Remote Playback, Buffer Scale Factor (1.3)

4.2.3 Experiment 3

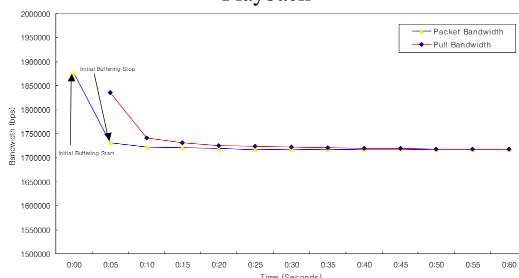
It is not easy to measure the overhead of the proposed buffer management scheme itself, due to the overhead of the media playback devices. Therefore, for accurate and reliable measurement of the overhead of the proposed scheme, we programmed two virtual media playback devices that support push and pull data transmission methods.

In Experiment 3, in order to estimate the performance overhead, the network packet bandwidth at the receiving buffer and the data transmission bandwidth at the virtual device were measured. The test media used in the experiment was MPEG data with attributes in Table 1, and 5-second Buffering Time and Buffer Scale Factor 1.3 were used.

Figure 12 shows the result of Experiment 3 (A) in which the media data was remotely played back using the pull-mode virtual device. In the experiment, there was no big difference between the *Packet Bandwidth* which is the receiving bandwidth of the media data packet and the *pull Bandwidth* at the media playback device. Since the *pull Bandwidth* was almost the same as the bitrate per second of the media data, we can say the overhead of the proposed scheme is negligible. In the experiment, the *push Bandwidth* was close to the bitrate per second of the stream. Also, there was little difference in bandwidth between pull and push mode.



[Figure 12] Experiment 3 (A): Pull, Remote Playback



[Figure 13] Experiment 3 (B): Pull, Local Playback

In Experiments 3 (B), the server and client programs have been executed on a local host, not in the remote playback mode on the network testbed. The results, again, showed that our scheme yielded negligible overhead regardless of the data passing mode of the buffer (pull or push) as shown in Figures 13.

5. Conclusion

In this paper, we have proposed an efficient and flexible push/pull buffer management mechanism for the client-side, which readily supports various media playback devices by providing a unified interface for both push and pull mode at the client's packet receiving buffer, and easily absorbs end-to-end network jitter. Moreover, the proposed scheme can save memory space compared to the case where a client keeps two types of buffers. We have implemented the scheme on the ISSA system and performed six experiments whose results showed good performance with little overhead.

We are planning to expand the proposed scheme to a buffer management scheme which works among a network server and clients, not just between the client's packet receiving buffer and media playback devices. Also, more study is needed to handle buffer overflow more efficiently without large overhead.

References

[1] M. Franklin, and S. Zdonik, "Data In Your Face: push Technology in Perspective", *In Proc. of the ACM SIGMOD International Conference on the Management of Data*, Vol. 27, No. 2, pp. 516-521, June, 1998.

[2] S. Rao, H. Vin, and A. Tarafdar, "Comparative Evaluation of Server-push and Client-push Architectures for Multimedia Servers", *In Proc. of the 6th International Workshop on Network and Operating System Support for Digital Audio and Video*, April, 1996.

[3] J. P. Martin-Flatin, "push vs. pull in Web-Based Network Management", *In Proc. of the Integrated Network Management VI*, pp. 3-18, May, 1999.

[4] C.G. Jeong, H.I. Kim, Y.R. Hong, E.J. Lim, S Lee, J. Lee, B.S Jeong, D.Y. Suh, K.D. Kang, John A. Stankovic, Sang H. Son, "Design for an Integrated Streaming Framework", *Department of Computer Science, University of Virginia Technical Report*, CS-99-30, November, 1999.

[5] Sun Microsystems, Java Media Framework API Guide, September, 1999, http://java.sun.com/products/java-media/jmf/index.html.

[6] S. Acharya, M. Franklin, and S. Zdonik, "Balancing push and pull for Data Broadcast", *In Proc. of ACM SIGMOD Conference*, Vol. 26, No. 2, pp. 183-198, May, 1997.

[7] Jack Y. B. Lee, "Concurrent push-A Scheduling Algorithm for push-Based Parallel Video Servers", *IEEE Transactions on Circuits and Systems for Video Technology*, Vol. 9, No. 3, pp. 467-477, April, 1999.

[8] H. Schulzrinne, S. Casner, R. Frederick, and V. /Jacobson, RTP: A Transport Protocol for Real-Time Applications, *IETF RFC 1889*, January 1996.

[9] Delivery System Architecture And Interfaces, *ftp://ftp.davic.org/Davic/Pub/Spec1_2/part04.pdf*