멀티미디어 통신 환경에서 Push/Pull 버퍼 관리 기법

정 찬 균[†] · 이 승 룡[††]

요 약

멀티미디어 통신 시스템은 하드웨어적으로 멀티미디어 데이터를 처리하기에 충분한 고성능의 컴퓨터 하드웨어와 고속 통신망 기술이 요구되며, 소프트웨어적으로는 멀티미디어의 특성상 대용량의 데이터를 처리하기 위한 버퍼의 사용을 효율적으로 관리하고 최적의 성능을 내기 위한 버퍼 관리 기법이 요구된다. 버퍼에 있는 데이터를 전달하는 방식에는 데이터를 제공하는 서버가 데이터 흐름을 제어하는 Push 방식과 데이터를 제공받는 클라이언트가 데이터 흐름을 제어하는 Pull 방식이 있으며, 이러한 데이터 전달 방식은 미디어 데이터를 수신하는 네트워크 클라이언트의 패킷 수신 버퍼와 수신된 미디어를 재생하는 미디어 재생 장치 사이에서도 적용된다. 하지만 기존의 멀티미디어 통신 시스템의 경우 클라이언트의 패킷 수신 버퍼를 위한 버퍼 관리 기법에서 미디어 재생 장치로의 데이터 공급을 위해 Push 방식과 Pull 방식 중의 하나만을 지원하여 다양한 미디어 재생 장치를 지원하기에는 한계가 있었으며, 두 가지 방식을 모두 지원한다 하더라도 단일화된 구조에서 제공하지 않아 사용하기 어렵다는 단점이 있다. 본 논문에서는 다양한 미디어 재생 장치로의 데이터 공급을 위해 단일 메모리에서 Push 방식과 Pull 방식을 모두 지원할 수 있고, 네트워크 지터의 흡수를 위해 버퍼링 기능을 제공할 수 있는 효율적이고 유연한 클라이언트에서의 Push/Pull 버퍼 관리 기법을 제안한다. 제안된 기법은 다양한 미디 재생장치의 각기 다른 버퍼관리 요구를 단일 메모리에서 일관되게 처리함으로써 메모리의 낭비를 방지할 수 있으며, 융통성 있고 단순한 버퍼관리가 가능하다. 모의 실험결과 제안한 버퍼 관리 기법은 기존에 개발된 통합기인 멀티미디어 통신 시스템 상에 적용되어 성능에 있어서 좋은 결과를 보였다.

A Scheme for Push/Pull Buffer Management in the Multimedia Communication Environments

Chan-Gyun Jeong[†] · Sungyoung Lee[††]

ABSTRACT

Multimedia communication systems require not only high-performance computer hardwares and high-speed networks, but also a buffer management mechanism to process many 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. Those buffering schemes can be applied to the data transfer between the packet receiving buffer, which receives media data from a network server, and media playout devices, which play the received media data. However, the buffer management mechanisms in client-sides mainly support either one of the Push or the Pull method. Consequently, they have some limitations to support various media playout devices. Futhermore, even though some of them support both methods, it is difficult to use since they can't provide a unified structure. To resolve these problems, in this paper, we propose an efficient and flexible Push/Pull buffer management mechanism at client-side. The proposed buffer management scheme supports both Push and Pull method to provide various media playout devices and to support buffering function to absorb network jitter. The proposed scheme can support the various media playback devices using a single buffer space which in consequence, saves memory space compared to the case that a client keeps two types of buffers. Moreover, it facilitates the single buffer as a mechanism for the absorbing network jitter effectively and efficiently. The proposed scheme has been implemented in an existing multimedia communication system, so called ISSA (Integrated Streaming Service Architecture), and it shows a good performance result compared to the conventional buffering methods in multimedia communication environments.

※ 과학재단 핵심전문 연구과제(과제번호 981-0923-124-2)
† 정 회 원 : 경희대학교 대학원 전자정보학부
†† 종신회원 : 경희대학교 전자정보학부
논문접수 : 1999년 12월 29일, 심사완료 : 2000년 2월 11일

1. 서 론

최근의 인터넷은 컴퓨터와 통신기술의 발전으로 인해 동영상, 음성 등과 같은 고대역폭을 요구하는 멀티미디어 데이터의 온라인 서비스를 가능하게 하였다. 이러한 멀티미디어 통신 서비스에는 VOD(Video On Demand), AOD(Audio On Demand), 화상회의(Video Conference), 인터넷 방송(Internet Broadcasting) 등과 같이 매우 다양한 서비스가 포함되며, 멀티미디어 데이터를 처리하기에 충분한 고성능의 컴퓨터 하드웨어와 고속 통신망 기술이 요구된다. 멀티미디어 통신 시스템은 대용량의 멀티미디어 데이터를 처리하기 위한 특성상 매우 큰 크기의 다양한 버퍼를 요구하며, 버퍼에 저장된 데이터에 대한 I/O 횟수가 상당히 많은 것이 특징이다. 따라서 버퍼에 대한 I/O를 효율적으로 처리하며, 응용 서비스 및 미디어에 알맞은 버퍼 크기를 계산하고, 버퍼 언더플로우와 버퍼 오버플로우에 효과적으로 대처할 수 있는 유연한 버퍼 관리 기법이 요구된다.

버퍼에 저장되어 있는 데이터의 전달 방식은 데이터 흐름의 주도권을 누가 가지고 데이터를 전달하는가에 따라 데이터를 공급하는 서버가 데이터 흐름을 제어하는 Server-Push(이하 Push로 기술함) 방식과 클라이언트가 데이터 흐름을 제어하는 Client-Pull(이하 Pull로 기술함) 방식으로 구분된다. Push 방식은 서버가 데이터 흐름의 주도권을 가지고 주기적으로 적절한 데이터를 클라이언트로 전송하는 방식으로서, 브로드캐스트 서비스에 적합하나 데이터를 공급하는 서버에서 데이터 전달 속도(bit-rate)를 적절하게 조정해야만 클라이언트에서의 버퍼 오버플로우 및 버퍼 언더플로우를 방지할 수 있다는 단점이 있다. 이와는 반대로 Pull 방식은 클라이언트가 데이터 흐름의 주도권을 가지고 서버에 원하는 데이터를 요청하면 서버는 요청된 데이터를 클라이언트로 전달하는 요청/응답 형식을 가지는 일종의 폴링 방식으로서, 일반적인 운영체제의 파일 시스템 I/O가 여기에 해당하며 클라이언트에서의 버퍼 오버플로우 및 버퍼 언더플로우를 클라이언트가 제어할 수 있으나, 유니캐스트 서비스에 적합하다는 제약이 있다[1, 2, 3]. 위와 같이 Push 방식과 Pull 방식은 각각 장단점이 있으며, 특히 멀티미디어 통신 시스템의 경우 네트워크 서버와 네트워크 클라이언트 사이뿐만 아니라 클라이언트의 패킷 수신 버퍼와 미디어 재생 장치 사이에서도 Push 방식 혹은 Pull 방식을 통해 버퍼에 있는 데이터가 미디어 재생 장치로 전달된다.

지금까지 멀티미디어 통신 시스템과 관련된 많은 버퍼 관리 기법에 관한 연구 진행되어 왔으나, 클라이언트에서의 버퍼 관리 기법은 종단간 네트워크 상황에 따른 버퍼 크기 제어 등과 같은 네트워크만을 고려한 버퍼 관리에 치우친 면이 많았으며, 대부분의 기법들이 클라이언트의 패킷 수신 버퍼와 미디어 재생 장치 간의 데이터 전달을 Push와 Pull 방식 중의 하나만 제공하였기 때문에 다양한 미디어 재생 장치를 지원할 수 없었다. 또한 Push와 Pull 방식을 모두 지원한다 하더라도 단일한 인터페이스로 제공하지 않아 다양한 미디어 재생 장치를 손쉽게 이용할 수 없다는 한계를 가지고 있다. 특히, ISSA (Integrated Streaming Services Architecture)[4, 5]와 같이 다양한 미디어들을 지원해야 하는 멀티미디어 통신 시스템에서 클라이언트의 패킷 수신 버퍼에서 미디어 별로 그에 알맞는 Push와 Pull 방식을 각기 따로 제공하는 경우는 버퍼관리 측면 비효율적이며, 메모리의 낭비를 피할 수 없다. 그리고, 인터넷과 같이 패킷 손실이 빈번하고 대역폭이 불안정한 네트워크 환경에서는 종단간 지터를 적응시키기 위한 버퍼링 기법도 요구된다.

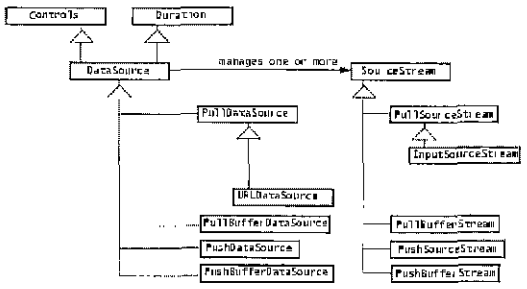
따라서 본 논문에서는 클라이언트에서의 패킷 수신 버퍼가 Push 방식과 Pull 방식의 데이터 출력을 단일한 인터페이스로 제공하여 다양한 미디어 재생 장치를 쉽게 지원할 수 있고, 버퍼링 기법을 제공하여 종단간 지터를 적응시킬 수 있는 효율적이고 유연한 클라이언트에서의 Push/Pull 버퍼 관리 기법을 제안한다. 제안된 버퍼링 기법은 다양한 미디어 장치마다 그에 알맞는 버퍼 passing을 위하여 stub와 같은 변환기를 설치해야하는 비효율적인 중복성을 피할 수 있었으며, 더 나아가 네트워크 지터를 적응하기 위한 버퍼링 기능도 같이 제공함으로써 버퍼링을 따로 수행하는 것에 비해 메모리 사용의 효율을 높일 수 있다.

본 논문의 구성은 다음과 같다. 제 2장에서는 Push/Pull 방식의 버퍼 관리 기법과 연관된 기존의 관련 연구를 살펴보고, 제 3장에서는 제안한 버퍼 관리 기법의 설계 구조를 제시하며, 제 4장에서는 제안한 버퍼 관리 기법의 구현과 결과를 설명하고, 마지막으로 제 5장에서 결론을 내린다.

2. 관련 연구

클라이언트에서의 버퍼 관리 기법과 관련된 연구는 매우 다양한 방면에서 많이 있었으나, 미디어 재생 장치를 위한 버퍼의 데이터 전달 방식과 연관된 Push와 Pull 방식의 버퍼 관리 기법에 관한 연구는 그리 많지 않다. 이번 장에서는 본 연구와 관련된 기존의 연구에 대해 살펴보겠다.

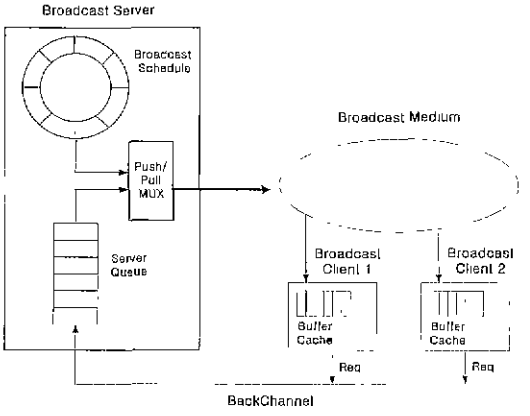
Sun Microsystems사가 연구 개발하고 있는 JMF(Java Media Framework)[6]에서는 *PushDataSource*와 *PullDataSource*라는 두 가지 인터페이스를 통해 각각 Push 방식과 Pull 방식을 위한 버퍼 기법을 제공한다. JMF는 멀티미디어 데이터 처리를 위한 단일한 구조와 프로그래밍 API를 제공하는 포괄적인 멀티미디어 플랫폼으로서, 자바를 이용한 네트워크 및 로컬에서의 멀티미디어 데이터의 재생 기능 등을 지원한다. 하지만 Push와 Pull 방식을 단일화된 인터페이스로 제공하지 않는 단점을 지니고 있어 실제 이를 이용한 프로그래밍 하기가 매우 까다롭다. (그림 1)은 JMF의 Push/Pull 데이터 소스 구조 모델을 보여준다.



(그림 1) JMF의 Push/Pull 데이터 소스 모델

Brown University에서는 Push 방식을 이용해 데이터를 브로드캐스트 서비스하는 시스템에서 Pull 방식을 함께 부가적으로 지원할 수 있는 방법과 그에 따른 버퍼 관리 기법을 제안하였다[7]. 이 연구에서 제안한 기법은 기존의 Push 방식의 브로드캐스트 시스템에 추가적으로 Pull 방식을 지원할 수 있는 서버와 클라이언트간의 독립적인 채널을 설정하고 이 채널을 통해 클라이언트의 요청을 전달함으로써 Pull 방식을 지원하였으며, 서버에 브로드캐스트 서비스 하고자 하는 데이터와 클라이언트에서 요청된 데이터를 멀티플렉싱하기 위한 Push/Pull MUX(Multiplexing) 버퍼를 두고 있다(그림 2). 그러나 이것은 네트워크 서버와 클라이언트 사이에서 Push 방식과 Pull 방식의 동시 지원에 초점을 맞춘 것으로서, 본 연구에서 제안한 기법과는 기본 개념이 다르며 여기서 Pull 방식은 단지 기존의 Push 방식을 통한 브로드캐스트 서비스의 단점을 보완하기 위해서 제공된다.

DAVIC(Digital Audio VIsual Council)[10]에서 제안된 버퍼관리 기법은 [7]에서 제시한 방법과 유사하나 백채널을 통해서 Pull 요구뿐만 아니라 클라이언트들이 자신들의 버퍼상태를 서버에게 알려서 미리 데이터를 받아두는 형태로, 서버와 클라이언트간의 버퍼 제어를 동적으로 수행하면서 네트워크의 지터를 적극적으로 흡수하는 방식이다. 그러나 이방법도 기본적으로 네트워크 서버와 클라이언트 사이에서 Push 방식과 Pull 방식의 동시 지원에 초점을 맞춘 것으로서 본 연구에서 제안하는 클라이언트에서 Push와 Pull 버퍼 passing을 들다 지원할 수 있는 통합된 구조와는 접근 방법이 다르다.



(그림 2) 브로드캐스트 서비스에서 Push/Pull 방식을 동시에 지원하는 구조

또한 Hong Kong University의 Jack Y. B. Lee는 Concurrent Push 방식을 사용한 병렬 비디오 서버 시스템에 관한 연구에서 서버 측에서의 버퍼 언더플로우와 클라이언트 측에서의 버퍼 오버플로우를 방지하기 위한 효율적인 버퍼 크기를 계산하는 버퍼 관리 기법을 제시하였다[8] 그러나 이 연구는 서버와 클라이언

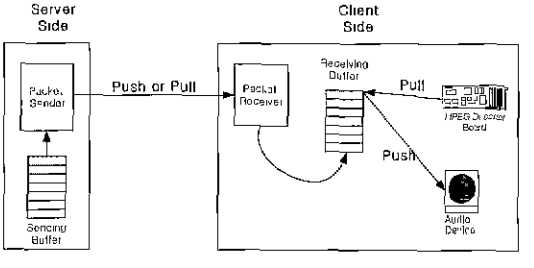
트간의 네트워크 전송에서 기존의 단일 전송대신에 병렬방식의 전송 기법을 사용함으로써, 매우 좋은 성능을 얻는 반면에 단일 전송에 비해 더 많은 크기의 버퍼 용량을 요구한다는 단점이 있으며, 본 연구에서 제안한 버퍼 관리 기법과는 개념이 다르다.

3. Push/Pull 버퍼 관리 기법의 설계

이 장에서는 제안한 클라이언트에서의 Push/Pull 버퍼 관리 기법의 동작 구조 및 버퍼 관리 기법을 위한 내부 버퍼 구조와 버퍼 관리를 위해 설계된 *InitBuffer*, *DeinitBuffer*, *StartBuffering*, *StopBuffering*, *AddData*, *PushData*, 그리고 *PullData*의 7개 알고리즘을 제시하고, 이에 대해 자세히 설명한다.

3.1 버퍼 관리 기법의 설계 모델

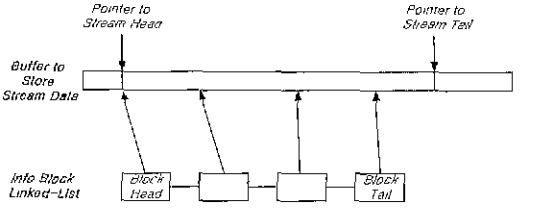
클라이언트에서의 Push/Pull 버퍼 관리 기법의 전체적인 동작 구조는 다음 (그림 3)과 같이 이루어지며, 실제 클라이언트에 위치한 Push/Pull 버퍼 관리 기법이 미디어 재생 장치와 연관되어 동작하는 구조를 표현한 것이다.



(그림 3) Push/Pull 버퍼 관리 기법의 동작 구조

(그림 3)에서 서버 측의 패킷 송신자를 통해 Push나 Pull 방식으로 전송된 패킷은 클라이언트에서의 패킷 수신자를 통해 수신되어 해석된 후 수신 버퍼로 전달되며, 수신 버퍼로 전달된 미디어 스트림 데이터는 설정된 미디어 재생 장치로 초기에 설정된 Push 혹은 Pull의 데이터 전달 방식을 이용해 재생된다. 즉 버퍼 관리 기법은 초기에 교섭을 통해 미디어 표현장치에 알맞은 데이터 전달 방식을 설정한 후 전송 세션이 시작하면 이미 설정된 전달 방식에 따라 데이터를 적절한 장치로 전달하며, 이 때 미디어 제생 장치는 설정된 데이터 전달 방식을 지원할 수 있는 장치이다. 또한 이때 버퍼로 데이터를 입력해주는 *Packet Receiver*는 서버로부터 데이터를 Push 방식으로 받을 수도 있으며 Pull 데이터 요청을 처리하는 stub을 설치하면 Pull 방식으로도 데이터를 받을 수 있어, 실제 수신 버퍼는 입력 데이터의 네트워크 전달 방식과 상관없이 유연하게 동작한다.

한편 제안한 버퍼 관리 기법은 Push와 Pull 방식을 단일의 버퍼에서 지원하고 Push와 Pull 방식의 모드 전환을 자유롭게 하기 위해 다음 (그림 4)와 같은 데이터 저장을 위한 내부 버퍼 구조를 가진다.



(그림 4) 내부 버퍼의 구조

(그림 4)에서 *Buffer to Store Stream Data*는 실제 네트워크를 통해 수신된 데이터가 저장되는 물리적인 선형 버퍼로서, 실제 저장된 데이터 스트림의 앞을 가리키는 *Pointer to Stream Head*와 데이터 스트림의 끝을 가리키는 *Pointer to Stream Tail*을 이용해 논리적으로 환형 버퍼로서 동작한다. *Info Block Linked-List*는 버퍼에 저장된 스트림을 블록 단위로 관리하여 블록마다 헤딩하는 정보를 저장하고 있는 정보 블록들의 연결 리스트로서, 버퍼가 Push 방식으로 동작할 때 필요한 구성요소이며, 각 블록은 블록 데이터가 표현되어야 하는 마감시간을 나타내는 *Expire Time*과 실제 데이터를 저장하는 선형 버퍼에서 블록이 표현하는 데이터를 위치를 가리키는 *Pointer to Data*, 그리고 블록이 표현하는 데이터의 크기를 나타내는 *Data Length*와 같은 정보를 저장하고 있다<표 1>.

<표 1> 버퍼 정보 블록의 내용

Expire Time	블록 데이터가 표현되어야 하는 마감시간
Pointer to Data	선형 버퍼에서 블록이 표현하는 데이터의 위치를 가리키는 포인터
Data Length	블록이 표현하는 데이터의 크기

3.2 버퍼 관리 기법의 알고리즘

제안한 버퍼 관리 기법을 위한 알고리즘은 다음 <표 2>에 나타난 것처럼 7개의 알고리즘으로 구성되며, 버퍼 초기화 및 소멸에 관련된 2 개의 알고리즘과 네트워크 지터 적응을 위한 버퍼링에 관련된 2개의 알고리즘, 그리고 버퍼에서의 데이터 입출력에 관련된 3개의 알고리즘으로 구분된다. 또한 각 알고리즘은 상호 보완적으로 동작하며, 각각의 동작 결과에 따라 서로 실행에 있어 영향을 받을 수 있다.

〈표 2〉 버퍼 관리 기법을 위한 알고리즘

InitBuffer	버퍼를 초기화하는 알고리즘
DeinitBuffer	버퍼를 소멸시키는 알고리즘
StartBuffering	버퍼링을 시작하는 알고리즘
StopBuffering	버퍼링을 중단하는 알고리즘
AddData	버퍼에 데이터를 추가하는 알고리즘
PushData	데이터를 Push 방식으로 전달하는 알고리즘
PullData	데이터를 Pull 방식으로 전달하는 알고리즘

다음 <표 3>은 제안한 버퍼 관리 기법의 알고리즘에서 사용되는 용어를 정리한 것으로서, *buffer passing mode*와 *buffering mode*는 버퍼에 관련된 상태를 표시하기 위한 것이고, *buffer data size*와 *playback delay time*은 버퍼와 관련한 통계 값이며, *buffering time*과 *bitrate of given media*, 그리고 *scale factor*는 버퍼링에 관련된 *buffering size*와 내부 버퍼의 크기인 *buffer size*를 계산하기 이용된다.

〈표 3〉 알고리즘에서 사용되는 용어

buffer passing mode	버퍼에 저장된 데이터 전달 모드로서 *Push* 혹은 *Pull* 의 값을 가짐
buffering mode	현재 버퍼링이 진행중인지 중단된 상태인지를 표시, *start*와 *stop*의 값을 가짐
buffering size	주어진 미디어 스트림을 위한 버퍼링 크기, 바이트 단위로서 다른 파라미터에 의해 계산됨
buffering time	주어진 미디어의 재생속도에 상대적인 값을 가지는 초 단위의 버퍼링 시간
bitrate of given media	주어진 미디어의 속도, 즉 초당 비트 속도
buffer size	buffering size에 scale factor를 곱해서 계산된 실제 내부 버퍼의 크기
scale factor	실제 이상적인 buffering size 만큼 내부 버퍼를 할당하지 않고 좀더 많은 크기의 버퍼를 할당하기 위한 요소
buffer data size	현재 동작중인 버퍼에 저장된 데이터의 바이트 단위의 크기를 의미
playback delay time	버퍼링 하는 동안의 지연등을 포함하는 총 재생 지연 시간을 의미

다음 (그림 5)는 버퍼를 초기화하는 *InitBuffer* 알고리즘의 의사코드로서 알고리즘 수행 초기에는 미디어 데이터를 표현할 미디어 표현장치에서 지원하는 버퍼 데이터 전달 방식(Push 혹은 Pull)을 알아내서 사용할 전달 방식을 결정한 후, 버퍼의 크기를 다음 공식과 같이 계산한다.

$$\textit{buffer size} = \textit{bitrate of the given media} * \textit{buffering time} * \textit{scale factor}$$

위의 공식에서 *bitrate of the given media*는 주어진 미디어의 초당 비트 속도이고, *buffering time*은 초 단위의 버퍼링 시간이며, *scale factor*는 이상적인 버퍼 크기보다 더 큰 실제 사용할 버퍼 크기를 구하기 위한 실수 값의 인자로서 *scale factor*가 1.0인 경우 계산된 버퍼의 크기는 이상적인 버퍼 크기와 동일하다. *scale factor*를 이용하여 버퍼의 크기를 이상적인 값보다 더 크게 해주는 이유는 서버로부터 미디어 데이터를 전달받는 수신 버퍼에서의 버퍼 오버플로우를 방지하기 위한 것이다. 한편 버퍼의 크기를 계산한 다음에는 실제 데이터를 저장할 메모리를 계산된 크기만큼 할당하고 초기 버퍼링 작업을 위한 *StartBuffering* 알고리즘을 호출한다.

```
InitBuffer Algorithm

1) Determine buffer passing mode for media device to use
   *) buffer passing mode
      = Push or Pull
2) Calculate buffer size
   *) buffer size
      = bitrate of the given media
        * buffering time * scale factor
3) Allocate memory for buffer
4) Invoke StartBuffering algorithm
```

〈그림 5〉 InitBuffer 알고리즘의 의사코드

DeinitBuffer 알고리즘은 *InitBuffer* 알고리즘과 반대로 버퍼를 소멸하는 알고리즘으로서 미디어 재생을 중단할 때 사용된다. (그림 6)은 *DeinitBuffer* 알고리즘의 의사코드로서 먼저 현재 버퍼링이 진행중이면 버퍼링 중단 알고리즘인 StopBuffering을 호출해서 버퍼링을

중단하고, 그 후 버퍼에 있는 데이터가 다 소모될 때까지 기다린 다음 버퍼를 위해 할당된 메모리를 해제한 후 종료한다.

```
DeinitBuffer Algorithm

1) If buffering mode == start then
   Invoke StopBuffering algorithm
2) Wait until all media stream data in
   buffer are consumed
3) Free memory for buffer
```

(그림 6) DeinitBuffer 알고리즘의 의사코드

버퍼링을 시작하는 *StartBuffering* 알고리즘은 처음에 버퍼링 상태를 *start*로 바꿈으로서 시작하며, 재생할 때 필요한 *playback delay time*을 계산하기 위해 현재 시간을 저장해 두고, 버퍼에 버퍼링 크기만큼의 데이터가 들어올 때까지 Push/Pull 오퍼레이션을 정지시킨 다음 그 조건을 만족하면 버퍼링을 중단시킨다(그림 7).

```
StartBuffering Algorithm

1) Change buffering mode to start
2) Save current time for calculation
   of playback delay time
3) 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
4) Invoke StopBuffering algorithm
```

(그림 7) StartBuffering 알고리즘의 의사코드

한편 버퍼링 크기는 버퍼 크기를 구하는 공식에서 *scale factor*를 곱하는 것을 제외한 공식을 이용해 구하며 다음과 같다.

$$buffering\ size = bitrate\ of\ the\ given\ media * buffering\ time$$

버퍼링을 중단하는 *StopBuffering* 알고리즘의 의사코드는 (그림 8)과 같으며, *StopBuffering* 알고리즘은 버퍼링 시작 알고리즘에서 버퍼링이 끝난 후에 호출되는 알고리즘으로서, 처음에 현재 시간과 버퍼링이 시작했던 때의 시간을 빼서 *additional playback delay time*을 구하고 이를 총 재생 지연 시간인 *playback delay time*에 더하며, 만약 현재 버퍼의 데이터 전달 모드가 *Push* 이면 다음 Push 방식의 데이터 전달 오퍼레이션을 위한 타이머 이벤트를 설정하고, 현재 버퍼의 데이터 전달 모드가 *Pull* 이면 잠들어 있는 Pull 방식의 데이터 전달 오퍼레이션을 깨우는 작업을 수행함으로써 Push/Pull 오퍼레이션의 잠금을 해제한다. 그런 후 버퍼링 상태를 *stop*으로 바꾸고 종료한다

```
Stop Buffering Algorithm

1) Update playback delay time by
 adding additional playback delay time
   *) additional playback delay time =
      current time - saved buffering start time
   *) playback delay time = additional
      playback delay time
2) Unlock Push/Pull operations
3) Change buffering mode to stop
```

(그림 8) StopBuffering 알고리즘의 의사코드

버퍼에 데이터를 추가하는 알고리즘인 *AddData*의 의사코드는 다음 (그림 9)와 같으며, 알고리즘 초기 시작에서 버퍼 오버플로우가 일어났는지 검사하여, 오버

```
AddData Algorithm

1) Lock buffer
2) Apply playback delay time to
   calculate expire time of new incoming data
3) If (size of new incoming data
   + buffer data size > buffer size)
   then
   Buffer overflow occurred
   Drop new incoming data
   Unlock buffer
   Terminate this algorithm
4) Copy new incoming data into buffer 5) Adjust
   pointer to stream tail with position of new data
6) If (buffer passing mode == Push)
   then Add a info block corresponding
   to new incoming data.
7) Update buffered data size
8) Unlock buffer
```

(그림 9) AddData 알고리즘의 의사코드

플로우가 일어났으면, 새로운 데이터를 포기하고 알고리즘의 실행을 중단한다. 만약 오버플로우가 일어나지 않았다면 새로 수신된 데이터를 버퍼로 복사하고, 현재 버퍼 데이터 전달 모드가 *Push*인 경우는 추가된 데이터에 헤딩하는 정보 블록을 정보 블록 링크드 리스트에 추가한다.

한편 버퍼 오버플로우가 일어난 경우의 처리는 현재 *AddData* 알고리즘 상에서 데이터를 포기하도록 설정되어 있어, 사용자 레벨의 서비스 품질(Quality of Services) 측면에서 바람직하지 않다. 따라서 이를 해결하기 위해 버퍼에 더 이상 추가할 수 없는 데이터를 위한 임시 버퍼를 만들어 여기에 임시로 저장해 두었다가 여분의 시간에 임시 버퍼에 있는 데이터를 실제 버퍼로 이동시키는 방안을 생각해 볼 수 있다. 하지만 이런 방식의 경우 임시 버퍼로 인한 버퍼 처리 오버헤드가 매우 커질 수 있기 때문에, 미디어 데이터의 실시간 처리를 보장할 수 없다는 단점이 있을 수 있어 이러한 오버헤드를 최소화하는 것이 관건이다.

다음 (그림 10)은 버퍼에 있는 데이터를 Push 방식으로 미디어 재생 장치로 출력하는 *PushData* 알고리즘의 의사코드를 보여주는 것으로서, 알고리즘 초기에 버퍼에 있는 데이터의 크기를 검사해 버퍼 언더플로우가 일어났다면, 버퍼링을 재시작하고 알고리즘을 중단하며, 버퍼 언더플로우가 일어나지 않은 경우는 미디어 재생 장치로 데이터를 Push하고 버퍼의 스트림 헤드에 대한 포인터를 바꿈으로써 버퍼에서 데이터를 삭제한 후, 다음 *PushData* 오피레이션을 위한 타이머 이벤트를 설정한다. 또한 Push 방식으로 동작하는 경우이므로 전달한 데이터에 해당하는 정보 블록도 함께 삭제한다.

```
PushData Algorithm

1) Lock buffer
2) If ( buffer data size < size of data to Push)
   then
        Buffer underflow occurred
        Invoke StartBuffering algorithm
        Terminate this algorithm
3) Push data of stream head in buffer to media device
4) Remove info block corresponding to removed data
5) Change pointer to stream head in buffer
6) Set up next timer event for Push Data algorithm
7) Unlock buffer
```

(그림 10) PushData 알고리즘의 의사코드

Pull 방식으로 미디어 재생 장치로 데이터를 출력하는 *PullData* 알고리즘의 의사코드는 다음 (그림 11)과 같다.

```
PullData Algorithm

1) Lock buffer
2) If ( buffer data size < size of data to Pull)
   then
        Buffer underflow occurred
        Invoke StartBuffering algorithm
        Sleep until buffering stop
3) Pull data of stream head in buffer to media device
4) Change pointer to stream head in buffer
5) Unlock buffer
```

(그림 11) PullData 알고리즘의 의사코드

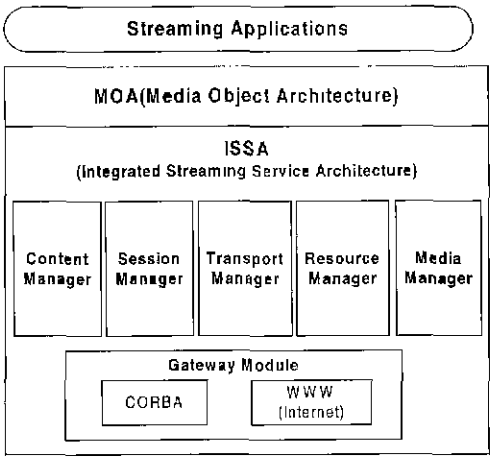
알고리즘의 초기에 버퍼 언더플로우를 검사해 만약 버퍼 언더플로우가 일어났다면, 버퍼링을 재시작하고 버퍼링이 멈출 때까지 기다린다. 이 것은 위의 *PushData* 알고리즘과는 크게 다른 점이라고 할 수 있다. 만약 버퍼 언더플로우가 일어나지 않았다면, 데이터를 Pull 방식으로 미디어 재생 장치로 전달하고, 전달한 데이터를 삭제한 후 종료한다. *PullData* 알고리즘은 *PushData* 알고리즘과는 다르게 타이머 이벤트를 사용하지 않고 미디어 표현 장치 인터페이스로부터 자율적으로 호출된다

4. 구현 및 성능평가

이 장에서는 본 논문에서 제안한 버퍼 관리 기법을 포괄적인 멀티미디어 시스템중의 하나인 통합 스트리밍 프레임워크인 ISSA에 적용하여 구현한 과정과 그 결과 및 실험에 대한 분석을 나타낸다

ISSA는 다양한 스트리밍 응용 프로그램을 위한 컨텐트 관리, 전송 프로토콜, 미디어 처리 등과 같은 여러 가지 기능을 제공하며, ISSA를 이용해 VOD, AOD 등과 같은 다양한 스트리밍 응용을 쉽게 제작할 수 있다. 다음 (그림 12)는 ISSA의 전체 구조를 나타내며, ISSA의 여러 구성요소 중에서 *Media Manager*는 MPEG, WAV, AU 등과 같은 저장된 미디어 파일의 처리 기능과 데이터베이스에 저장된 미디어 스트림 처리 기능, 미디어 데이터를 다른 형식으로 인코딩하고 디코딩하기 위한 A/V 코덱 기능, 미디어를 표현하기

위한 오디오/비디오 장치 제어 기능 등과 같은 다양한 미디어 처리 기능을 제공한다.



(그림 12) ISSA의 전체 구조

*Media Manger*는 또한 크게 *Media Source*와 *Media Sink*로 구성되며 *Media Sink*는 클라이언트에서 전송된 미디어 스트림 데이터를 받아 이를 적절한 미디어 표현 장치로 공급하는 역할을 수행하므로, 제안한 버퍼 관리 기법은 *Media Sink* 안에 포함 되여 동작하도록 구현하였다. 또한 ISSA 내부에 구현된 버퍼 관리 기법의 성능을 측정하기 위해 파일로 저장된 미디어 스트림 데이터를 전송하는 간단한 서버 프로그램과 서버로부터 전송된 미디어 스트림 데이터를 재생하는 간단한 클라이언트 프로그램을 개발하였다(그림 13).

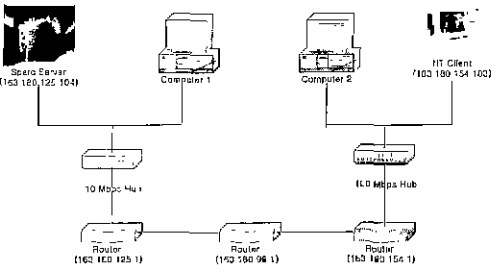
(그림 13) GUI 클라이언트 프로그램의 동작 화면

구현된 서버 프로그램은 Microsoft Windows-NT와 Sun Solaris 2.X 환경에서 동작하며, 클라이언트 프로그램은 Microsoft Windows NT 환경에서 동작한다. 서버와 클라이언트간의 미디어 데이터 전송은 IETF (Internet Engineering Task Force)에서 RFC (Request For Comments) 문서로 제안한 RTP (Real-time Transport Protocol)[9]를 사용하여 미디어의 비트율에 따라 Push 방식으로 이루어지며, 동작하는 네트워크 환경은 TCP/IP 기반의 10/100 Mbps Ethernet 환경이고, 프로그램은 C++ 언어를 이용해 구현하였다<표 4>.

<표 4> 서버와 클라이언트 프로그램의 동작환경

구분 / 항목	서버	클라이언트
운영체제	Windows NT or Solaris 2.X	Windows NT
전송 프로토콜	RTP(Real-time Transport Protocol)	
네트워크	TCP/IP over 10/100 Mbps Ethernet	
개발언어	C++ Language	

*RTP*는 *Henning Schulzrinne* 등에 의해 RFC 1889로 제안된 멀티미디어 데이터의 실시간 전송을 지원하는 프로토콜로서, 현재 버전 2가 나와있으며 오디오/비디오와 같이 실시간 특성을 가진 멀티미디어 데이터의 종단간 실시간 전달 서비스를 제공하며 구현된 프로그램에서는 ISSA *Transport Manager*에서 제공하는 UDP(User Datagram Protocol) 위에서 동작하는 RTP 루틴을 이용하였다.



(그림 14) 테스트베드의 구성도

(그림 14)는 제안한 Push/Pull 버퍼 관리 기법의 성능을 측정하기 위해 구축한 테스트베드(testbed)의 구성도로서 서버 프로그램은 IP 주소가 163.180.125.104인 Sun Ultra SPARC 서버 머신에서 동작하며, 클라이언

트 프로그램은 IP 주소가 163.180.154.183인 Windows NT를 사용하는 Intel Pentium II 클라이언트 머신에서 동작한다. 또한 서버 머신과 클라이언트 머신 사이에는 세 개의 라우터가 있으며, 서버 머신이 있는 곳의 네트워크 환경은 10 Mbps Ethernet이며 클라이언트 머신이 속한 곳의 네트워크 환경은 100 Mbps Ethernet이다.

구현된 Push/Pull 버퍼 관리 기법의 기능과 성능을 평가하고 오버헤드를 측정하기 위해서 작성된 서버 프로그램과 클라이언트 프로그램을 이용해서 총 6번의 실험을 수행하였다. 처음 두 번의 실험에서는 버퍼링 및 Push 방식과 Pull 방식의 데이터 전달 기능이 제대로 동작하는지를 검사하였고, 다음 네 번의 실험에서는 버퍼 관리 기법 자체의 오버헤드를 측정하기 위한 결과를 살펴보았다. 먼저 실험 1은 MPEG-1 System 스트림을 네트워크를 통해 재생한 경우로서, 이 때 버퍼 관리 기법은 Pull 방식을 사용하여 전달된 MPEG 데이터를 DirectShow 미디어 표현 장치로 보내게 된다. 실험 1에서 사용한 미디어 스트림 데이터의 속성은 다음 <표 5>와 같다.

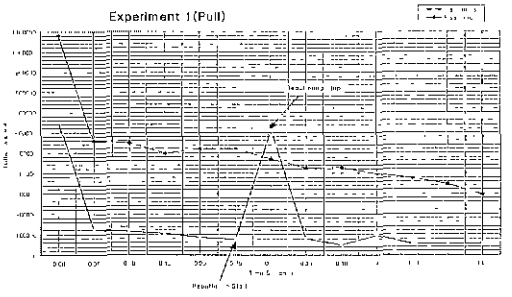
〈표 5〉 실험 1에서의 테스트 미디어 속성

Encoding Type	MPEG-1 System Layer Stream
Bitrate	1,715,200 bps
Resolution	320x240
Total Stream Length	50.4 MBytes

〈표 6〉 실험 1에서의 버퍼 속성값

Buffering Time	3 seconds	5 seconds
Buffering Size	1,715,200 bits / 8 * Buffering Time = 643,200 bytes	1,715,200 bits / 8 Buffering Time = 1,072,000 bytes
Buffer Scale Factor	1.3	1.3
Buffer Size	Buffering Size * Buffer Scale Factor = 836,160 bytes	Buffering Size * Buffer Scale Factor = 1,393,600 bytes

실험 1에서 버퍼링 시간을 3초와 5초로 다르게 준 경우 계산된 버퍼링 크기와 버퍼의 크기는 각각 다음 <표 6>과 같았으며, 버피링 시간을 3초로 준 경우에는 (그림 15)와 같이 네트워크 패킷 지연으로 인한 리버퍼링(rebuffering)이 1분 동안의 재생 시간동안 1번 있었으나, 버피링 시간을 5초로 길게 준 경우에는 리버퍼링(rebuffering)이 일어나지는 않았고 버피에 있는 데이터의 크기가 계속 감소하였다. 이는 서버와 클라이언트간의 네트워크 전송 지연에 따른 것으로서 특히 MPEG-1 System Laver 스트림과 같이 고대역폭을 요구하는 미디어 스트림의 경우 전송 지연 시간이 매우 길어 질 수 있다.



(그림 15) 실험 1 Pull 방식인 경우 buffer data size의 통계

실험 2에서는 Push 방식을 테스트하기 위한 실험으로서, 서버에서 PCM(Pulse Code Modulation) 오디오 형식으로 저장된 AU 파일을 전송하면 클라이언트의 버퍼 관리 기법은 Push 방식을 통해 오디오 장치로 전송된 미디어 데이터를 전달하며, 오디오 장치를 전달된 데이터를 재생하는 실험이었다. 한편 실험 2에서 사용된 .au 파일 형식의 PCM 오디오 파일의 미디어 속성은 다음 <표 7>과 같다.

〈표 7〉 실험 2에서의 테스트 미디어 속성

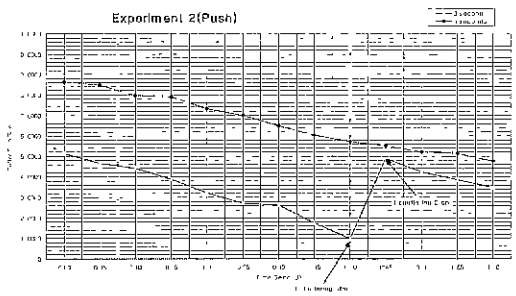
Encoding Type	Linear PCM Audio
Bitrate	1,411,200 bps
Sampling Rate	44,100 Hz
Number of Channels	2
Bits Per Sample	16

실험 2도 실험 1과 마찬가지로 버퍼링 시간을 3초와 5초로 다르게 준 경우 계산된 버퍼링 크기와 버피의 크기는 각각 다음 <표 8>과 같았으며, (그림 16)은 1분간 재생하는 동안 버퍼에 저장된 미디어 데이터 크기의 변화 추이를 보여준다. Push 방식으로 PCM 오디오 데이터를 오디오 장치로 출력하는 작업에서도 역시 버퍼링 시간이 3초인 경우에는 중간에 버퍼 언더플로우가 발생하여 재생 시간 동안 리버피링이 1번 있었

으나, 5초인 경우에는 버퍼링 시간이 긴 관계로 리버퍼링이 발생하지 않았다. 따라서 제안한 버퍼 관리 기법은 위의 Pull 방식으로 동작할 때와 Push 방식으로 동작할 때 모두 버퍼의 데이터 전달 모드에 상관없이 버퍼 언더플로우가 발생한 경우 이를 해결하기 위한 네트워크 버퍼링 기능이 동작하여 네트워크 패킷 전송 지터를 적응시킬 수 있다는 것을 알 수 있다.

〈표 8〉 실험 2에서의 버퍼 속성값

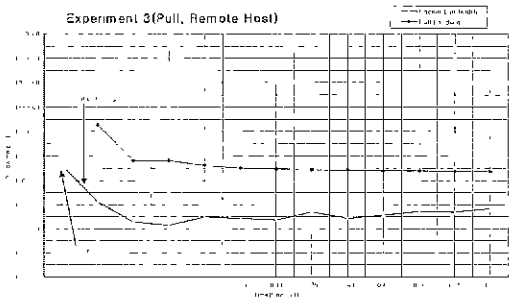
Buffering Time	3 seconds	5 seconds
Buffering Size	1,411,200 bits / 8 Buffering Time = 529,200 bytes	1,411,200 bits / 8 Buffering Time = 882,000 bytes
Buffer Scale Factor	13	13
Buffer Size	Buffering Size * Buffer Scale Factor = 687,960 bytes	Buffering Size * Buffer Scale Factor = 1,146,600 bytes



(그림 16) 실험 2 : Push 방식인 경우 buffer data size의 통계

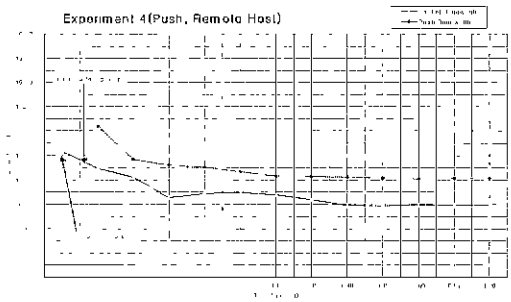
한편 위의 실험에서는 제안한 버퍼 관리 기법을 실제 미디어 재생 장치에 적용하여 그 기능이 제대로 동작하는지 평가할 수 있었으나, 미디어 재생 장치에서의 디코딩 및 재생 등의 오버헤드로 인하여 버퍼 관리 기법 자체만의 오버헤드를 측정하는 것이 어렵다. 따라서 버퍼 관리 기법의 오버헤드를 정확히 측정하기 위해 Push 방식과 Pull 방식의 데이터 전달을 지원하는 두 개의 소프트웨어적인 가상적인 미디어 재생 장치를 만들어, 실제 미디어 재생 장치를 이용하지 않고 가상적인 장치를 이용하는 네 번의 실험을 통해 버퍼 관리 기법의 오버헤드를 간접적으로 측정하였다.

실험 3과 실험 4는 각각 Pull 방식과 Push 방식의 가상 장치를 이용한 경우로서 버퍼 관리 기법의 오버헤드를 추론할 수 있는 네트워크 수신 버퍼에서의 패킷 수신 대역폭과 가상 장치에서의 데이터 전달 대역폭을 측정하였으며, 이 때 전송한 미디어 스트림은 실험 1에서 사용한 미디어와 동일하다. 다음 (그림 17)은 Pull 방식의 가상 장치를 이용한 실험으로서 미디어 데이터 패킷의 수신 대역폭인 *Packet Bandwidth*와 미디어 재생 장치로 Pull 방식으로 전달되는 데이터의 대역폭인 *Pull Bandwidth*의 차이가 없었으며, *Pull Bandwidth*의 값이 미디어 데이터의 초당 비트 속도에 거의 일치함으로써 버퍼 관리 기법에서의 오버헤드는 거의 없었다.



(그림 17) 실험 3 : Pull 방식, 원격호스트

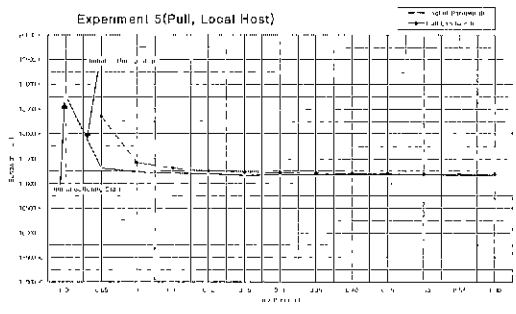
(그림 18)은 Push 방식의 가상 장치를 이용한 경우로서 여기서도 역시 *Push Bandwidth*의 값이 미디어 데이터의 초당 비트 속도에 근접하였으며, *Pull Bandwidth*와의 성능의 차이를 찾아 볼 수 없었다.



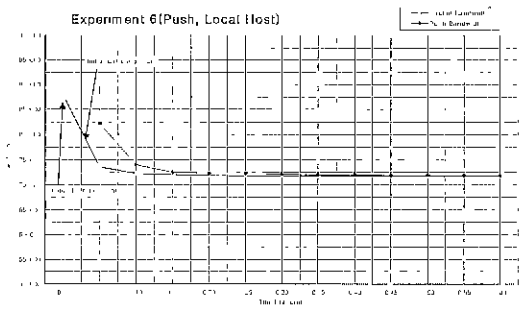
(그림 18) 실험 4: Push 방식, 원격호스트

마지막으로 실험 5와 실험 6은 실험 3과 실험 4와는 다르게 (그림 14)에서의 데스트베드 환경이 아닌 로컬 호스트에서 서버 프로그램과 클라이언트 프로그램을

동시에 동작시켜 얻은 결과로서, 이 경우도 역시 Pull 방식과 Push 방식 모두 별다른 오버헤드 없이 정상적으로 동작하였다



(그림 19) 실험 5 : Pull 방식, 로컬 호스트



(그림 20) 실험 6 Push 방식, 로컬 호스트

(그림 19)와 (그림 20)은 실험 5와 실험 6에 대한 결과 값을 보여준다.

5. 결 론

멀티미디어 시스템은 그 특성상 대용량의 데이터를 처리하기 위해 상당히 큰 크기의 버퍼를 사용하며, 버퍼 사용을 효율적으로 관리하기 위한 버퍼 관리 기법이 요구된다. 특히 멀티미디어 통신 시스템은 종단간 지터를 적용시키고, Push 방식과 Pull 방식의 데이터 전달 방식을 모두 제공하여 다양한 미디어 재생 장치를 지원할 수 있는 일관적이고 효율적인 클라이언트에서의 버퍼 관리 기법이 요구된다.

하지만 지금까지 연구된 멀티미디어 통신 시스템을 위한 클라이언트에서의 버퍼 관리 기법은 종단간 네트워크 상황에 따른 버퍼 크기 제어 등과 같은 네트워크만을 고려한 버퍼 관리에 치우친 면이 많았으며, 대부분의 기법들이 클라이언트의 패킷 수신 버퍼와 미디어 재생 장치간의 데이터 전달을 Push 방식과 Pull 방식 중의 하나만 제공하거나 두 방식을 모두 제공한다 하더라도 단일한 인터페이스로 제공하지 않아 다양한 미디어 재생 장치를 손쉽게 이용할 수 없다는 한계를 가지고 있다

따라서 본 논문에서는 멀티미디어 통신 시스템에서 클라이언트의 버퍼 관리 기법이 일관된 인터페이스를 통해 Push 방식과 Pull 방식의 출력을 모두 지원하여 다양한 미디어 재생 장치를 지원하고, 네트워크 패킷 버퍼링 기법을 사용하여 종단간 지터를 적응시키는 효율적이고 유연한 클라이언트에서의 Push/Pull 버퍼 관리 기법을 제안하였으며, 제안한 기법을 포괄적인 멀티미디어 통신 시스템인 ISSA 시스템 상에서 구현하여 적은 오버헤드를 가지고 좋은 성능을 보였다

그러나 제안한 버퍼 관리 기법은 현재 클라이언트의 패킷 수신 버퍼와 미디어 재생 장치 사이에서의 버퍼 제어만 다루고 있어서, 이를 네트워크 서버와 네트워크 클라이언트 사이에서도 동작할 수 있는 버퍼 관리 기법으로 확장시킬 필요가 있다 또한 버퍼 오버플로우에 효율적으로 대처하기 위한 오버헤드가 적은 방안에 대한 구체적인 연구가 요구된다

참 고 문 헌

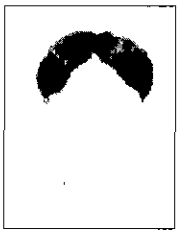
[1] M. Franklin, and S Zdonik. "Data In Your Face: Push Technology in Perspective," In Proc. of the ACM SIGMOD International Conference on the Management of Data, Vol.27, No.2, pp 516-521, June, 1998.

[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] 정찬균, 이승룡, "통합 스트리밍 프레임워크의 설계", 제 11회 한국 정보처리학회 춘계 학술발표 논문집,

pp.319-322, 1999년 4월.
[5] 정찬균, 김형일, 홍영래, 임익진, 이승제, 이승룡, 정병수, "분산 스트리밍 시스템 설계", 제4회 한국 멀티미디어 학회 추계 학술발표 논문집, pp.338-343, 1999년 11월.
[6] Sun Microsystems. Java Media Framework API Guide, September, 1999. http://java.sun.com/products/java-media/jmf/index.html.
[7] 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.
[8] 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.
[9] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, RTP : A Transport Protocol for Real-Time Applications, IETF RFC 1889, January 1996
[10] Delivery System Architecture And Interfaces, ftp://ftp.davic.org/Davic/Pub/Spec1_2/part04.pdf



정 찬 균

e-mail : cgjeong@oslab.kyunghee.ac.kr
1998년 경희대학교 전자계산공학과 졸업(학사)
2000년 경희대학교 대학원 전자계산공학과(공학석사)
2000년~현재 (주)에스큐브 근무
관심분야 실시간시스템, 멀티미디어시스템

이 승 룡

e-mail : sylee@oslab.kyunghee.ac.kr
1978년 고려대학교 재료공학과 학사
1986년 Illinois Institute of Technology 전산학과 석사
1991년 Illinois Institute of Technology 전산학과 박사
1992년~1993년 Governors State University 조교수
1993년~현재 경희대학교 전자계산공학과 부교수
관심분야 : 실시간 시스템, 실시간 고장허용시스템, 멀티미디어 시스템.