커널 레벨 RTP를 지원하는 확장 BSD 소켓 API
(Extended BSD Socket API Supporting Kernel-level RTP)

최 문 선 [†] 김 경 산 [†] 김 성 조 [††]
(Mun Seon Choi) (Kyung San Kim) (Sung Jo Kim)

요 약 유무선 통신 기술 및 인터넷의 발전으로 인터넷 방송, VOD 등과 같은 멀티미디어 서비스가 일반화되고 있다. RTP는 인터넷상에서 실시간 멀티미디어 데이타를 전송하는데 적합하도록 IETF에서 제정한 프로토콜이다. RTP는 주로 라이브러리 형태로 구현되어 다양한 애플리케이션에 사용되나, 라이브러리 형태로 사용되는 RTP는 성능 측면에서 문제가 있어 이를 개선한 프로토콜이 embeddedRTP이다. 본 논문에서는 기존의 커널 레벨 RTP인 embeddedRTP를 기반으로 그 문제점을 보완하며, API를 네트워크 애플리케이션에서 널리 사용되는 BSD 소켓 API와 통합하고 그 성능을 개선한 ExtendedERTP를 제안한다. embeddedRTP의 API가 BSD 소켓 API에 통합되어 기존의 네트워크 스택이 RTP를 내장하게 되면, 사용자들은 별도의 RTP 라이브러리를 사용할 필요 없이 익숙한 BSD 소켓 API 형태의 인터페이스를 통해 실시간 데이타를 송수신할 수 있다. 본 논문은 또한 embeddedRTP에 비해 패킷 처리 속도를 15~20% 가량 향상시키는 방안과 패킷 처리에 필요한 메모리 요구량을 기존의 3.5% 수준으로 줄일 수 있는 방안을 제시한다.

키워드 : RTP, 멀티미디어 서비스, 실시간 서비스

Abstract Due to the evolution of wired and wireless communication technologies and the Internet, multimedia services such as Internet broadcast and VOD have been prevalent recently. RTP is designed to be suitable for transmission of real-time multimedia data on the Internet by IETF. While a variety of applications have utilized different RTPs implemented as a library, embeddedRTP is RTP-based kernel-level protocol that resolved performance issues of this kind of RTPs. This paper proposes the ExtendedERTP protocol based on existing embeddedRTP. This new protocol resolves a couple of issues such as packet processing overhead and buffer requirement and combines its APIs with BSD socket APIs which have been widely utilized in network applications. This paper demonstrates that this integration makes it possible to transmit real-time multimedia data through the accustomed interface of BSD socket APIs with nominal extra overhead. This paper also proposes a scheme for improving packet processing time by 15~20% and another scheme for reducing memory requirement for packet processing to about 3.5%, comparing with those of embeddedRTP.

Key words : RTP, Multimedia Service, Real-time Service

1. 서 론

최근 인터넷의 트래픽은 동영상 스트리밍 서비스와 같은 실시간 멀티미디어 데이타의 특성을 갖는 경우가 많다. 그러나 인터넷은 본질적으로 단순히 텍스트 또는 그래픽의 처리에 적합하도록 설계되어 있어 멀티미디어 데이타를 전송하는데 적합하지 않다. 이를 보완하여 실시간 데이타를 지원함으로써 인터넷에서 멀티미디어 서비스가 가능하도록 개발된 프로토콜이 RTP(Real-time Transport Protocol)[1]이다.

인터넷에서 실시간 트래픽이 급속도로 증가함에 따라 RTP는 중요한 프로토콜로서 자리매김하고 있으며, RTP 전송속도와 QoS 향상에 대한 연구가 널리 진행되고 있다[2,3]. 현재 RTP의 구현은 대부분 라이브러리 형태로 이루어지고 있고, 각 애플리케이션마다 RTP 라이브러리를 독자적으로 구현하는 예가 많기 때문에, 동일한 시스템에서 동작하는 각 애플리케이션마다 별도의

· 본 연구는 정보통신부 및 정보통신연구진흥원의 대학 IT연구센터 (홈네트워크연구센터) 지원사업의 연구결과로 수행되었음

† 정 회 원 : 중앙대학교 컴퓨터공학과
windchoi@konan.cse.cau.ac.kr
mountain@konan.cse.cau.ac.kr
†† 종신회원 : 중앙대학교 컴퓨터공학과 교수
sjkim@cau.ac.kr

논문접수 : 2005년 4월 25일
심사완료 : 2006년 2월 27일

RTP 라이브러리를 중복적으로 사용하는 문제점이 있다. 최근에 RTP를 커널 레벨에서 구현하여 이러한 문제를 해결하고 그 성능을 향상시키기 위한 SRTPIO[4], embeddedRTP[5]와 같은 연구가 진행되었다. 그러나, 이러한 연구들은 RTP 자체에 대한 연구가 아니거나, 메모리 요구량 및 데이타 복사 횟수, 그리고 Blocking I/O를 지원하지 못하는 등의 문제점을 가지고 있을 뿐 아니라, RTP를 사용하기 위해서 독자적으로 정의된 시스템 콜 사용하고 있어, 그 사용법을 따로 배워야 하는 번거로움이 있다.

이에 본 논문에서는 기존의 커널 레벨 RTP인 embeddedRTP를 기반으로 Blocking I/O 기능을 추가하고, 참조버퍼를 구현하여 메모리 복사 및 메모리 요구량을 최소화하여 성능을 개선하였으며, API를 네트워크 애플리케이션에서 널리 사용되는 BSD 소켓 API와 통합한 ExtendedERTP(Extended EmbeddedRTP)를 제안한다. embeddedRTP의 API가 BSD 소켓 API에 통합되어 기존의 네트워크 스택이 RTP를 내장하게 되면, 사용자들은 별도의 RTP 라이브러리를 사용할 필요 없이 익숙한 BSD 소켓 API 형태의 인터페이스를 통해 실시간 데이타를 송수신할 수 있다.

본 논문의 구성은 다음과 같다. 2장에서는 RTP 프로토콜의 동작원리와 그 문제점에 대해서 살펴본 후, 3장에서는 BSD 소켓 API로의 통합 방안 및 embedded-RTP의 성능 개선에 대해서 기술한다. 4장에서는 BSD 소켓 API로의 통합으로 인해 발생하는 오버헤드를 측정하고, 본 논문에서 구현한 ExtendedERTP와 embedded-RTP와의 성능을 비교한 후, 마지막으로 5장에서는 결론을 맺는다.

2. RTP 프로토콜

2.1 동작 원리

RTP는 기본적으로 UDP를 기반으로 순차 전송과 흐름제어를 수행하여 종단간 실시간 데이타 전송을 담당하지만, QoS를 위한 트래픽의 제어는 수행하지 않는다. 그러므로 이를 보완하기 위해서 RTP와 함께 사용되는 프로토콜이 RTCP(RTP Control Protocol)이다. 또한 RTP를 사용하여 멀티미디어 데이타를 전송하고자 할 때, 세션의 제어와 미디어 스트림을 동기화하기 위한 상위 프로토콜로서 RTSP(Real Time Streaming Protocol) [6], H.323[7], SIP(Session Initiation Protocol)[8] 등이 주로 사용된다.

RTP 프로토콜의 동작 시나리오는 크게 RTP 세션의 생성, RTP 패킷의 전송, RTP 패킷의 수신, RTP 세션의 해제 등의 단계로 나눠질 수 있다. RTP 프로토콜은 세션 생성을 위한 정보 교환과 데이타 송수신 제어 등을

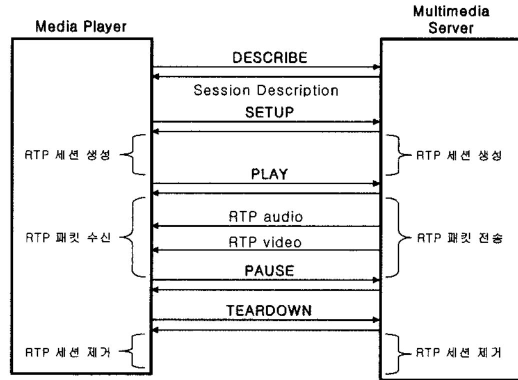


그림 1 RTSP를 사용할 때의 RTP의 동작 시나리오

위해 RTSP와 같은 별도의 상위 프로토콜을 이용한다.

그림 1은 대표적인 상위 프로토콜인 RTSP 프로토콜에서 RTP가 어떻게 동작하는지를 보여주고 있다. 먼저, 클라이언트는 서버에 RTSP의 DESCRIBE 메시지를 전송함으로써 해당 서버에 대한 정보와 미디어에 대한 프로파일 정보 등을 얻어 온 다음, RTSP의 SETUP 메시지를 교환하여 송수신 포트 및 각 RTP 세션에 대해 RTP 스트림의 소스를 식별하는 SSRC(Synchronization Source)를 비롯한 전송 방법을 설정한다. 서버와 클라이언트는 DESCRIBE와 SETUP 메시지를 통해서 교환한 정보를 바탕으로 각각 RTP 세션을 생성하게 된다. RTP 세션 생성이 완료되면 PLAY 메시지를 교환한 다음, 멀티미디어 데이타의 송수신이 이루어진다. 필요한 데이타 송수신이 끝나고 클라이언트에서 TEARDOWN 메시지를 보내면, 서버는 이에 대한 응답을 보내고 양쪽 모두 생성된 RTP 세션을 제거한다.

표 1은 그림 1의 RTP 프로토콜의 동작 시나리오에서 각 단계별로 수행되는 항목을 나타내고 있다. 보통 하나의 수행 항목은 하나의 라이브러리 함수로 처리되지만, 하나의 라이브러리 함수가 몇 개의 과정을 통합하여 처리하기도 한다.

2.2 RTP의 문제점

RTP는 주로 라이브러리 형태로 개발되어 사용되는 사례가 많은데, 그 대표적인 예로는 UCL RTP[9], oRTP [10], RTPlib[11] 등이 있다. 표 2는 표 1의 RTP 동작 과정 항목을 수행하는 API의 이름을 RTP 라이브러리별로 정리한 것이다.

표 2에서 볼 수 있듯이 RTP 라이브러리들은 동일한 동작을 수행하는데 각각 다른 사용자 인터페이스를 제공하고 있다. 이는 RTP를 사용하는 애플리케이션을 작성하는 개발자들로 하여금 혼란을 일으킬 뿐 아니라, 다른 RTP 라이브러리를 사용하기 위해서는 새롭게 해당 라이브러리의 함수 이름과 사용법을 배워야 하는 번거

표 1 RTP의 동작 과정

단계	항목번호	항목
RTP 세션 생성 및 연결 설정	1	RTP 세션을 생성하고 초기화한다
	2	RTP 패킷을 전송할 송신주소 및 송신포트를 설정한다
	3	RTP 패킷을 수신할 수신주소 및 수신포트를 설정한다
	4	RTP 및 RTCP용 소켓을 생성하고 바인드한다
	5	RTP 세션에서 사용할 SSRC 값을 설정한다
RTP 패킷의 전송	1	전송할 데이타에 RTP 헤더를 붙이고 RTP 패킷을 생성한 후 송신 주소 및 포트로 전송한다
RTP 패킷의 수신	1	수신된 RTP 패킷을 읽어온다
연결 해제 및 RTP 세션 제거	1	상대방에게 RTCP BYE 메시지를 보낸다
	2	RTP 및 RTCP용 소켓을 제거한다
	3	RTP 세션 관련 자료구조를 제거하고 RTP 세션을 제거한다

표 2 RTP 라이브러리의 API

단계 \ RTP	항목 번호	RTPlib	oRTP	UCL RTP
RTP 세션 생성 및 연결 설정	1	RTPCreate	rtp_session_new	rtp_init
	2	RTPSessionSetLocalAddr	rtp_session_set_local_addr	rtp_init
	3	RTPSessionSetReceiveAddr	rtp_session_set_local_addr	rtp_init
	4	RTPOpenConnection	rtp_session_set_remote_addr	rtp_init
	5	RTPMemberInfoSetSSRC	rtp_session_set_ssrc	rtp_set_my_ssrc
RTP 패킷의 전송	1	RTPSend	rtp_session_send_with_ts	rtp_send_data
RTP 패킷의 수신	1	RTPReceive	rtp_session_recv_with_ts	rtp_recv_data
연결 해제 및 RTP 세션 제거	1	RTPCloseConnection	rtp_session_destroy	rtp_send_bye
	2	RTPCloseConnection	rtp_session_destroy	rtp_done
	3	RTPDestroy	rtp_session_destroy	rtp_done

로움이 있다. 또한, 각기 다른 RTP 라이브러리를 사용하고 있는 애플리케이션들이 동일한 시스템에서 동작할 경우, RTP 프로토콜 스택이 중복 구현되어야 하는 문제가 있다. 이 구조는 리소스의 낭비를 초래할 뿐 아니라 성능 면에서 좋지 않다.

최근에 진행된 커널 레벨에서 실시간 데이타를 지원하기 위한 연구로는 Kernel-level SCTP[12,13], Real-Time Audio Server[14], Linux DVB API[15] 등을 들 수 있다. Kernel-level SCTP는 SCTP를 커널에 이식하여 멀티미디어 스트리밍을 지원하고자 한 연구로서, SCTP는 RTP와는 달리 IP 상위의 TCP/UDP와 동일한 전송계층의 프로토콜로 TCP/UDP와의 공존여부, SCTP 킬러 애플리케이션 여부 등에서 그 활용에 의문이 제기되고 있다. Real-Time Audio Server에 대한 연구는 오디오 버퍼의 수준에 따라 오디오 인코딩 레벨을 달리하고 실시간 데이타를 지원하는 스케줄링을 기법을 사용하여, 커널 레벨에서 실시간을 지원하는 오디오 스트리밍 서버를 구현하고자 하는 연구로서, 실시간 프로토콜 자체에 대한 연구는 아니다. Linux DVB API는 리눅스 커널 레벨에서 Video 및 Audio 등을 위한 API를 정의한 것으로 BSD 소켓 API와 다른 자체적인 API들로 구성되어 있다.

또한, 최근에 RTP를 커널 레벨에서 구현하여 그 성능을 향상시키기 위한 SRTPIO[4], embeddedRTP[5]와 같은 연구가 진행되었다. SRTPIO(Special RTP I/O)는 RTP가 주로 멀티미디어 파일 전송에 이용되는 것을 고려하여 커널 수준에서 특수한 파일 I/O 모듈만을 구현한 것으로 RTP 자체에 대한 연구로는 볼 수 없다. embeddedRTP는 RTP 프로토콜 스택을 커널 수준에서 구현한 것으로 문맥 교환(Context Switching) 및 동적 메모리 할당 등의 오버헤드를 최소화하고 임베디드 시스템에 적합하도록 설계되었다. 그러나, embeddedRTP는 다음과 같은 측면에서 문제점을 가지고 있다.

- 이 프로토콜이 제공하는 함수의 호출을 위해 독자적으로 정의된 시스템 콜을 사용하고 있어, 애플리케이션 개발자가 새로운 시스템 콜 인터페이스를 습득해야 하는 불편함이 있다.
- 네트워크 디바이스 드라이버에서 패킷을 수신하는 sk_buff 공간, RTP 계층에서 패킷을 복사하는 RTP 버퍼 공간, 그리고 커널과 사용자 공간 사이의 복사를 위해 필요한 공간까지 3개의 버퍼 메모리를 사용하고 있다. 이는 하나의 패킷에 대해 2번의 복사와 중복으로 메모리 공간을 요구하는 문제점이 있다.
- 스트리밍 데이타의 송신 기능을 제공하지 못하고 수신 기능만을 제공하고, Non-Blocking 수신만을 제공하는 등 완전한 I/O를 제공하지 못하는 문제점이 있다.

이에 본 논문은 기존의 커널 레벨 RTP인 embeddedRTP를 기반으로 그 문제점을 보완하며, API를 네트워크 애플리케이션에서 널리 사용되는 BSD 소켓 API와 통합하고 그 성능을 개선한다. 그림 2는 RTP 스택이 BSD 소켓 인터페이스를 통하여 커널 네트워크 스택에 포함되었을 때의 구조를 나타낸 것이다. 애플리케이션은 기존의 BSD 소켓 API을 통하여 커널 내부에 존재하는 RTP를 사용하고, RTP는 하위 프로토콜인 UDP를 통하여 데이타를 전송한다. 이 구조에서 애플리케이션은 별도로 RTP 라이브러리를 포함하지 않아도 되며, 애플리케이션 개발자는 익숙한 BSD 소켓 인터페이스를 사용하여 손쉽게 RTP를 사용할 수 있는 장점이 있다.

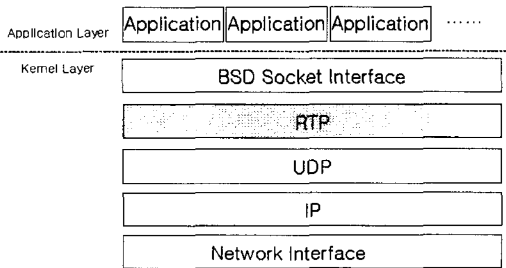


그림 2 BSD 소켓을 통해 커널에서 수행되는 RTP 스택의 구조

3. ExtendedERTP

3.1 ExtendedERTP가 포함된 커널의 구조

그림 3은 본 논문에서 구현한 ExtendedERTP가 커널의 BSD 소켓 API를 사용할 때의 구조를 나타낸 것이다. 기존의 BSD 소켓은 BSD 소켓 계층과 INET 소켓 계층을 통하여 TCP/IP 및 UDP/IP로 연결되는 구조를 가지고 있다. 일반적으로 RTP는 TCP의 재전송 기능이 실시간 전송에는 적합하지 않으므로 체크섬과 멀티플렉싱 기능만을 활용할 수 있도록 UDP를 전송 프로토콜로 사용하며, 그림 3에서 보는 바와 같이 UDP의 상위 프로토콜로 위치하게 된다. 일반적으로, 애플리케이션이 TCP나 UDP를 사용할 때는 기존과 동일한 구조로 통신이 이루어지고, RTP를 사용할 때만 하위 프로토콜로 UDP를 이용하여 통신하게 된다.

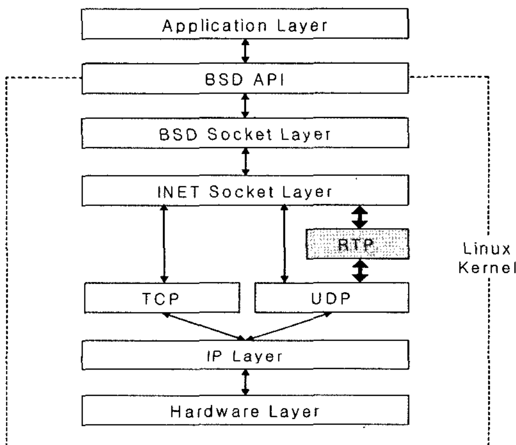


그림 3 RTP를 지원하는 BSD 소켓의 구조

3.2 참조 버퍼의 구현

RTP를 이용하여 데이타를 수신할 때 계속적으로 수신되는 RTP 패킷을 관리하기 위해 RTP 패킷 큐를 사용한다. RTP 패킷이 송신될 때는 순서 번호에 따라 순차적으로 전송되지만, 수신될 때는 인터넷 환경의 특성상 순서대로 도착하지 않을 수 있다. RTP 패킷 큐는 계속적으로 수신되는 RTP 패킷을 버퍼링하고, 패킷의 순서번호에 맞게 정렬하기 위해서 사용된다. RTP가 라이브러리 계층에서 구현된 경우에는 애플리케이션이 이를 직접 처리하면 되나, 커널 수준에서 구현된 경우는 커널 내부적으로 RTP 패킷 큐를 생성하여 RTP 패킷을 관리해야 한다.

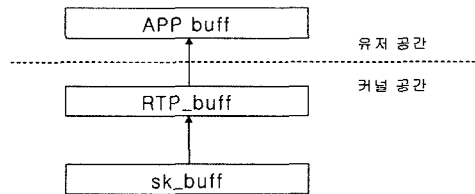


그림 4 RTP 패킷의 버퍼 구조

그림 4는 embeddedRTP에서 RTP 패킷 큐를 관리하기 위해서 필요한 버퍼 구조를 보여준다. 우선, 네트워크 디바이스를 통해서 수신된 패킷은 네트워트 소켓 버퍼에 저장되는데, 수신된 패킷이 RTP 패킷이면, RTP 모듈은 소켓 버퍼(sk_buff)로부터 RTP 패킷을 RTP 버퍼(RTP_buff)로 읽어와서 해당 RTP 세션의 RTP 패킷 큐에 순서 번호에 맞추어 저장하게 된다. 애플리케이션은 커널 내부의 RTP 패킷 큐에서 차례대로 RTP 패킷을 유저 공간에 있는 자신의 버퍼(APP_buff)로 읽어오게 된다. 이러한 구조는 2번의 메모리 복사와 메모리 공간을 중복 요구하는 문제점이 있다.

애플리케이션 계층의 버퍼와 RTP 패킷 큐에 있는 RTP 패킷 버퍼는 사용자 공간과 커널 공간에 각각 존재하므로 불가피하게 메모리 복사가 발생한다. 그러나, RTP 버퍼와 소켓 버퍼는 같은 커널 공간에 존재하므로 성능 및 메모리 요구량의 최적화를 위해서 복사 횟수를 최소화해야 한다.

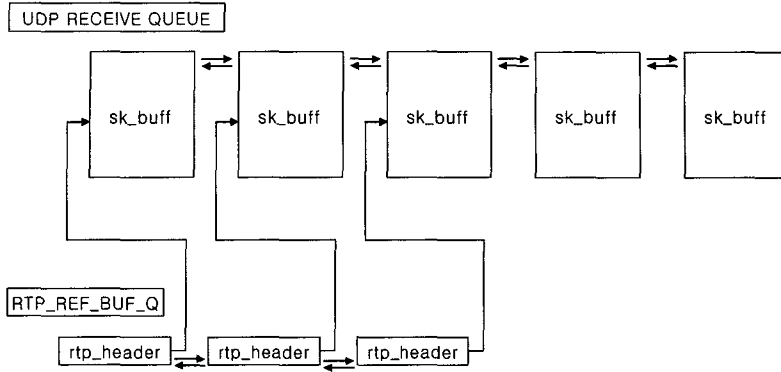


그림 5 ExtendedERTP의 참조 버퍼

그림 5는 ExtendedERTP에서 구현한 참조 버퍼의 구조를 나타내고 있다. 참조 버퍼는 소켓 버퍼에 저장된 전체 RTP 패킷을 RTP 패킷 버퍼에 복사하는 것이 아니라, RTP 헤더 정보만을 복사하고 페이로드는 소켓 버퍼의 위치를 참조하게 하는 방법으로 데이타 복사 오버헤드를 최소화하였다. 이 구조는 RTP 패킷 큐에 저장된 RTP 패킷의 개수가 많아질수록 메모리 요구량 면에서 많은 이득이 기대된다.

3.3 BSD 소켓 API로의 통합

ExtendedERTP는 BSD 소켓 API를 통하여 RTP를 사용할 수 있도록 기존의 BSD 소켓 API를 수정하였다. 우선 RTP 세션을 관리하기 위해 사용되는 RTP 세션 구조체의 포인터 변수를 BSD의 소켓 구조체에 추가하였다. RTP 세션 구조체에도 BSD 소켓 구조체에 대한 포인터를 추가하여 BSD 소켓 구조체와 RTP 세션 구조체간의 상호 참조가 가능하도록 하였다. 또한, BSD 소켓 구조체에 RTP를 위한 소켓임을 나타낼 수 있는 rtp 필드를 추가하였다. BSD 소켓 API의 각 함수들은 BSD 소켓 구조체의 rtp 필드를 검사하여 이 필드가 설정되어 있으면 RTP 관련 함수를 호출하고 그렇지 않으면 기존의 함수를 호출하는 구조를 가진다.

표 3은 상위 프로토콜로서 RTSP가 사용될 때 BSD 소켓 API를 통하여 ExtendERTP가 사용되는 예를 보여주고 있다. 표 3에서 사용된 API의 사용 방식은 기존의 네트워크 프로그래밍 스타일과 동일하다. 우선, RTP 세션 생성 과정은 RTP 세션 관련 구조체 생성, RTP 및 RTCP용 소켓 생성, 수신 주소 및 포트 설정, 송신 주소 및 포트 설정 등의 과정으로 이루어진다. 이러한 과정은 BSD 소켓의 socket, bind, connect 시스템 콜을 통하여 수행된다.

- socket 시스템콜 : RTP 세션 관련 구조체와 RTP 및 RTCP용 소켓을 생성한다.
- bind 시스템콜 : 세션 관련 구조체에 수신 주소와 포트를 설정하고, RTP 및 RTCP용 소켓을 수신 주소와 포트에 바인드한다.
- connect 시스템 콜 : 세션 관련 구조체에 송신 주소와 포트를 설정하고, RTP 및 RTCP용 소켓을 송신 주소와 포트에 연결한다.

표 3 BSD 소켓 API를 통하여 ExtendedERTP가 사용되는 예

```
/* RTP 세션 생성 */
rtpfd = socket(AF_INET, SOCK_RTP, 0);
bind(rtpfd, (struct sockaddr *)&clientaddr, sizeof(clientaddr));
connect(rtpfd, (struct sockaddr *)&serveraddr, sizeof(serveraddr));
getsockopt(rtpfd, IPPROTO_RTP, RTP_RTCPFD, &rtcpfd, &len);        // optional
setsockopt(rfpfd, IPPROTO_RTP, RTP_SSRC, &ssrc, &len);

/* 데이타 송수신 */
write(rtpfd, (char *)buf, len); // 멀티미디어 서버일 경우
  또는
read(rtpfd, (char *)buf, len);  // 미디어 플레이어일 경우

/* RTP 세션 제거 */
close(rtpfd);
```

표 3의 RTP 세션 생성 부분에서 보는 바와 같이, BSD의 socket 시스템 콜을 사용하여 RTP 세션을 생성하기 위해 RTP용 소켓 타입 상수인 SOCK_RTP를 추가로 정의한다. 또한, RTCP 메시지는 커널 내부에서 적절한 주기에 따라 전송되기 때문에 보통의 경우 애플리케이션이 직접 전송하지 않는다. 하지만, 애플리케이션이 RTCP의 APP 패킷을 독자적으로 정의하여 전송하고자 한다면 RTCP 파일 디스크립터가 필요하게 되는데, 이 경우 BSD의 getsockopt 시스템 콜을 이용하여 얻을 수 있다. 세션에서 사용할 SSRC와 같은 옵션 값을 설정하기 위해서 BSD의 setsockopt 시스템콜이 사용된다. setsockopt/getsockopt 시스템 콜의 level 인자는 RTP용으로 IPPROTO_RTP를 추가로 정의하고, optname 인자는 필요에 따라 새로운 상수값을 정의하여 사용한다. 추가적으로 RTP 세션에서 사용할 대역폭, RTP 패킷의 TTL 값, 그리고, RTCP 패킷 중 SSRC, CNAME 등과 같은 송신자의 정보를 전달하는 SDES (Source DEScription) 패킷에서 사용할 정보 등을 설정한다.

RTP 세션 생성 후에 표 3의 데이타 송신 부분에서 보는 바와 같이, socket 시스템 콜에서 반환된 RTP 소켓의 파일 디스크립터를 사용하여 데이타 스트림을 전송하거나 수신한다. 멀티미디어 서버는 write/send/sendto 시스템 콜을 사용하여 데이타를 송신하게 되며 미디어 플레이어는 read/recv/recvfrom 시스템 콜을 사용하여 데이타를 수신하게 된다. 데이타의 송수신이 끝나면, 표 3의 RTP 세션 제거 부분에서 보는 바와 같이, 데이타의 전송 및 수신이 끝나면 close 시스템콜을 호출하여 사용된 RTP 세션 및 파일 디스크립터를 제거한다.

RTP 패킷의 데이타 전송 품질에 대해 피드백을 제공함으로써 애플리케이션 계층에서 QoS가 보장될 수 있도록 본 논문에서는 다음과 같은 방법으로 RTCP를 사용한다. 즉, RTCP 패킷이 도착하면 커널의 RTP 모듈은 도착한 RTCP 메시지의 정보를 버퍼에 저장하고, 애플리케이션 계층에 사용자 정의 시그널(SIGUSR1)을 전송하여 이를 알린다. RTCP 메시지의 내용이 필요한 경우, 애플리케이션은 getsockopt 시스템 콜을 이용하여 그 내용을 참조한다. 또한 SDES 메시지의 CNAME, NAME, EMAIL 등의 값을 설정할 때에는 setsockopt 시스템 콜을 이용한다.

3.3.1 socket 시스템 콜

그림 6은 socket 시스템 콜에서 RTP를 지원하기 위해 변경된 구조를 나타내고 있다. 원래의 socket 시스템 콜은 내부적으로 sys_socket 함수와 연결되고, sock_create 함수와 sock_map_fd 함수를 각각 호출한다. sock_create 함수는 소켓 구조체를 생성하는 역할을 하고, sock_map_fd 함수는 생성된 구조체에 하나의 파일 디스크립터를 매핑시키는 역할을 한다.

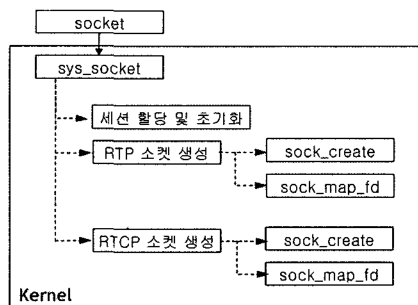


그림 6 socket 시스템 콜의 변경

RTP를 사용하기 위해서 호출된 socket 시스템 콜에서는 RTP 세션 할당이 이루어져야 하고, RTP와 RTCP 용으로 두개의 소켓이 생성되어야 한다. 그러므로, socket 시스템 콜은 소켓 타입 상수를 체크하여 SOCK_RTP이면 우선, RTP 세션을 할당하여 RTP 세션을 관리할 자료구조 생성 및 초기화 작업을 수행하도록 변경된다. 그 다음, 소켓 구조체를 할당하고 파일 디스크립터를 맵핑시키는 sock_create 및 sock_map_fd 함수의 호출 부분이 RTP와 RTCP를 위해 두 번 반복되도록 변경된다. 마지막으로, RTP 및 RTCP용으로 소켓을 각각 생성하고, 파일 디스크립터를 할당받으면 이를 RTP 세션 관리 자료구조에 저장하며, 소켓 구조체와 세션 구조체를 연결한 후, RTP 파일 디스크립터를 리턴하도록 수정된다. RTP가 아닌 TCP 및 UDP용 소켓을 생성할 경우에는 RTP 소켓을 위한 조건문만을 검사할 뿐, 기존의 루틴대로 수행된다.

3.3.2 bind 및 connect 시스템 콜

그림 7과 그림 8은 bind 및 connect 시스템 콜에서 RTP를 지원하기 위해 변경된 구조를 나타내고 있다. 우선, bind 시스템 콜은 내부적으로 sys_bind 함수와 연결되고, sys_bind 함수에서 inet_bind 함수를 호출하여 bind 작업을 수행한다. RTP를 지원하기 위해 bind 시스템 콜에서는 RTP 및 RTCP에서 사용할 수신 주소 및 포트를 RTP 세션 관련 자료구조에 저장하고 이것을 소켓에 바인드하는 작업이 수행된다. 즉, bind 시스템 콜은 인자로 들어온 파일 디스크립터를 이용하여 해당 소켓 구조체의 rtp 필드를 검사하여 RTP를 위한 소켓이면, 인자로 들어온 수신 주소와 포트번호를 RTP 관련 자료구조에 저장하고, 이것을 RTP 소켓에 바인드한다. 그 다음, 수신 주소는 동일하게 하고 포트번호만 1 증가시켜 RTCP 소켓에 대해서도 동일한 작업을 수행한다.

connect 시스템 콜은 RTP 및 RTCP에서 사용할 송

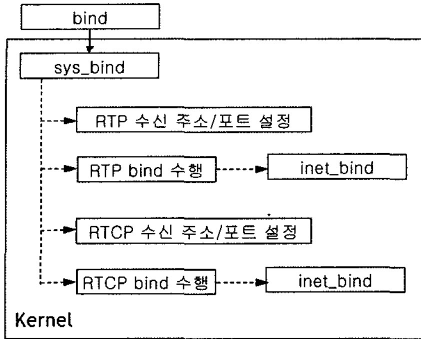


그림 7 bind 시스템 콜의 변경

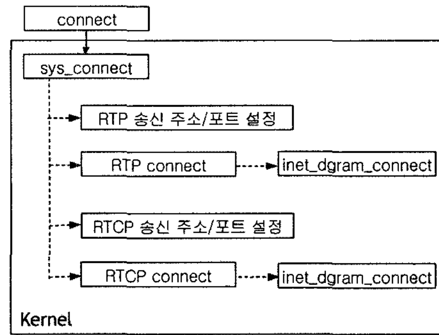


그림 8 connect 시스템 콜의 변경

신 주소 및 포트를 RTP 세션 관련 자료구조에 저장하고 이것을 소켓에 연결한다. 즉, RTP를 위한 소켓이면, 인자로 들어온 수신 주소와 포트를 RTP 관련 자료구조에 저장하고, 이것을 소켓에 연결한다. 그 다음, RTCP 소켓에 대해서도 bind 시스템 콜에서와 마찬가지 방식으로 연결 작업이 수행된다.

3.3.3 write 시스템 콜

write 시스템 콜은 그림 9와 같은 구조를 가진다. write 시스템 콜은 커널 내부의 sys_write 함수와 연결되고, sys_write 함수는 네트워크 통신을 위한 소켓의 경우에 sock_write 함수와 연결된다. sock_write 함수는 하위 계층의 sock_sendmsg, inet_sendmsg, udp_sendmsg 함수를 차례로 호출하여 데이타를 송신한다. 애플리케이션이 페이로드를 커널로 내려보내면, RTP 모듈은 RTP 헤더정보를 페이로드 앞에 첨가하여 하위 프로토콜 스택을 통해 전송한다. 또한, RTP 패킷 전송 후, RTCP 패킷 중 데이타 송신에 대한 품질 및 통계 정보를 알리는 데 사용되는 SR(Sender Report) 패킷을 위한 통계 정보 갱신 작업이 수행된다.

RTP를 위한 소켓인지 여부를 검사할 때, 소켓 구조체에 접근할 수 있는 sock_write 함수에서 검사가 이루어진다. RTP를 위한 소켓이면, RTP 헤더 정보를 설정하여 RTP 패킷을 전송한 다음, 전송한 총 패킷수와 총 바이트 수와 같은 RTCP 정보를 갱신한다. send/sendo 시스템 콜은 내부적으로 write 시스템 콜과 유사한 구조를 가지므로 write 시스템 콜과 마찬가지 방법으로 변경된다.

3.3.4 read 시스템 콜

read 시스템 콜은 write 시스템 콜과 유사한 구조를 가지고 있다. read 시스템 콜은 커널 내부의 sys_read 함수와 연결되고, 소켓의 경우에 sys_read 함수는 sock_read 함수와 연결된다. sock_read 함수는 하위 계층의 sock_recvmsg, inet_recvmsg, udp_recvmsg 함수를 차례로 호출한다.

그림 10은 read 시스템 콜에서 RTP 패킷을 수신하기 위해 변경된 구조를 나타내고 있다. read 시스템 콜에서도 write 시스템 콜과 마찬가지로, sock_read 함수에서 소켓 구조체의 rtp 필드를 검사하여 RTP를 위한 소켓인지를 조사하도록 변경된다. RTP를 위한 소켓이면 해당 세션의 RTP 패킷 큐로부터 RTP 패킷을 읽어와서 리턴하게 된다.

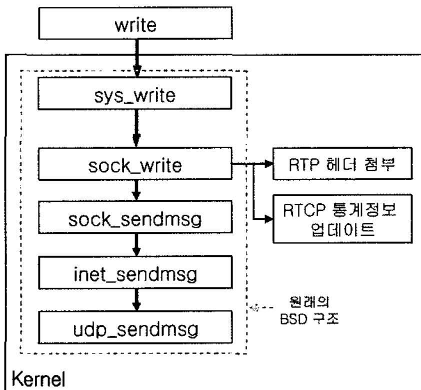


그림 9 write 시스템 콜의 변경

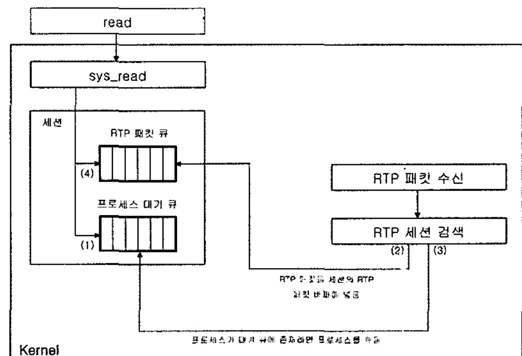


그림 10 변경된 read 시스템 콜의 구조

또한, UDP 계층으로 패킷이 수신되고 RTP 세션이 존재하면, RTP 모듈은 해당 세션들의 포트번호를 검사하여 RTP 패킷인지를 검사하게 된다. RTP 패킷이면, 소켓 버퍼에서 RTP 헤더만을 읽어와서 RTP 패킷 큐에 순서번호를 고려하여 삽입된다. recv/recvfrom 시스템 콜은 내부적으로 read 시스템 콜과 유사한 구조를 가지므로 read 시스템 콜과 마찬가지 방법으로 변경된다.

BSD의 데이타 송수신 시스템 콜은 기본적으로 Blocking/Non-Blocking 송수신을 지원한다. 예를 들어, write, read 시스템 콜은 fcntl 시스템 콜을 사용하여 송수신 모드를 설정하고, send/sendto, recv/recvfrom 시스템 콜은 인자에 MSG_DONTWAIT 플래그 설정하는 방법으로 송수신 모드를 설정한다. 이것은 RTP를 이용할 때도 동일하게 적용되는데, embedddeRTP에서는 Non-Blocking 모드만을 지원할 뿐 Blocking 모드를 지원하지 못하는 문제점이 있었다. ExtendedERTP는 Blocking/Non-Blocking 송수신을 모두 지원한다. 송신의 경우는 기존의 방식과 동일하게 사용할 수 있지만, 수신의 경우는 기존 BSD 소켓 API 내부의 하위 계층의 함수를 호출하여 데이타를 수신하는 것이 아니라 RTP가 자체적으로 관리하는 RTP 패킷 큐를 이용하므로, 별도로 Blocking 수신을 지원할 수 있는 메커니즘이 필요하다.

그림 10의 (1)~(4) 과정은 read 함수에서 RTP 패킷을 Blocking 모드로 데이타를 수신하기 위한 구조를 보여주고 있다. Blocking 수신 모드일 때, sock_read 함수는 RTP 패킷 큐에 읽을 데이타가 없으면 에러코드를 리턴하는 대신, 사용자 프로세스를 대기큐에 추가하고 sleep 시킨다(그림 10의 (1)). 사용자 프로세스가 sleep 하고 있는 동안 RTP 패킷이 수신되면, 어떤 RTP 세션의 패킷인지를 조사한 후, 소켓 버퍼로부터 패킷을 수신하여 해당되는 세션의 RTP 패킷 버퍼로 RTP 패킷을 저장한다(그림 10의 (2)). RTP 패킷을 RTP 패킷 버퍼에 저장할 때마다 해당 세션의 대기큐에 프로세스가 존재하는지를 검사하게 되는데, 대기큐에 프로세스가 존재하면 해당 프로세스를 깨운다(그림 10의 (3)). 깨어난 프로세스는 자신을 대기큐에서 제거하고, RTP 패킷 버퍼를 검사하여 저장된 패킷이 있으면 데이타를 읽은 후에 리턴한다(그림 10의 (4)).

3.3.5 close 시스템 콜

그림 11은 close 시스템 콜의 수행 과정을 나타낸 것이다. close 시스템 콜은 내부적으로 sys_close 함수에 연결되며, sys_close 함수는 파일이 아닌 네트워크 통신을 위한 소켓을 닫고자 하는 경우에는 sock_close에 연결된다.

RTP는 세션을 제거하기 전에 BYE RTCP 메시지를

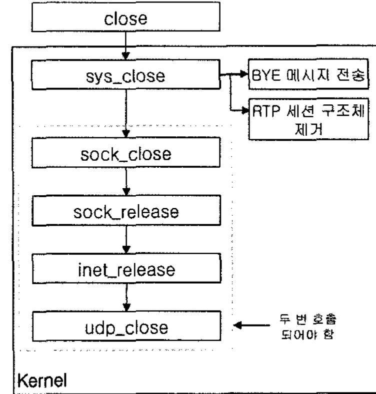


그림 11 close 시스템콜의 변경

보낸다. 그러므로, RTP를 위한 소켓이면, sys_close 시스템 콜은 BYE RTCP 메시지를 전송한다. 그 다음, 소켓을 제거하는 루틴을 두 번 호출하여 RTP 및 RTCP 소켓을 제거한 후, 마지막으로 RTP 세션 관련 구조체를 제거한다.

4. 성능 분석

ExtendedERTP의 성능을 다음과 같은 방법으로 평가한다. 우선, ExtendedERTP의 API를 BSD 소켓 API로 통합함으로 인해 발생하는 BSD 소켓 API의 오버헤드를 측정한다. 그 다음, embeddedRTP와 Extended-ERTP의 성능을 각각 측정하여 비교한다. 성능측정을 위해 MPEG4IP 프로젝트[16]의 MP4LIVE와 다윈 스트리밍 서버(Darwin Streaming Server)[17]가 스트리밍 서버로서 사용되었으며, 표 4와 같은 사양의 데스크톱 PC 상에서 실행되었다. 또한, 미디어 플레이어로서는 MPEG4IP 프로젝트의 MPEG4IP 플레이어가 사용되었으며, 표 5와 같은 사양의 PDA를 이용하여 리눅스 기반의 PDA용 운영체제인 Familiar 0.7.1상에서 실행되었다.

4.1 BSD 소켓 API 오버헤드

앞서 설명한 바와 같이 embeddedRTP API를 기존 BSD 소켓 API와 통합하기 위해서는 부득이하게 기존 BSD 소켓 API의 내부 모듈이 다음과 같이 변경되었다. 즉, 기존의 수행 과정에 RTP 소켓인지를 검사하는 조건문이 첨가된다. 기존의 TCP/UDP 소켓인 경우에는 조건에 대한 검사만을 수행하고 기존의 코드를 그대로 수행하게 되고, RTP 소켓일 경우에만 첨가된 조건문 내의 RTP 관련 코드를 수행하도록 되어 있다. 표 6에서와 같이 BSD 소켓 시스템 콜은 해당 시스템 콜에 대한 동작을 수행하기 전에 소켓 구조체의 rtp 필드를 통해 RTP 소켓인지를 검사한 후, RTP 소켓이면 해당 시스템 콜에서 RTP 소켓에 대해 필요한 처리를 수행하게 된다.

기존의 TCP/UDP 소켓 사용의 경우와 비교할 때, 본 논문에서 수정된 리눅스 커널의 BSD 소켓 API에 첨가되는 조건문의 횟수는 다음과 같다.

표 4 PC 사양

H/W	CPU	Intel Pentium4 2.6GHz
	RAM	256Mbyte
	NIC	3Com
S/W	OS	Linux
	커널버전	2.4.20

표 5 PDA 사양

H/W	모델명	Compaq iPAQ H-3970
	CPU	Intel Xscale PXA 250
	RAM	64 MB
	Flash 메모리	48 MB
	NIC	Compaq WL100
S/W	Familiar 0.7.1(Kernel 2.4.19)	

표 6 BSD 시스템 콜에서 RTP 소켓의 처리

```
......
if (sock->rtp)
{
        /* RTP 소켓에 대한 처리 */
}
......
```

- socket 시스템 콜 : 3회
- bind 및 connect 시스템 콜 : 2회
- read 및 write 시스템 콜 : 1회
- close 시스템 콜 : 4회

표 7은 표 4의 환경에서 위에 열거한 시스템 콜들에 첨가된 조건문 코드의 수행시간을 측정한 것이다. 한번의 조건문 수행시간을 ns 단위까지 측정하는 것은 어렵기 때문에, 각 조건문을 10만 번 반복한 수행시간을 측정하여 한번의 조건문 수행시간을 계산하였다. 수정된 시스템콜 중에서 가장 많은 조건문 검사가 추가된 close 시스템 콜의 조건문 수행시간이 5.38 ns 정도를 보이고 있으며, 첨가된 조건문의 횟수가 가장 적은 read 및 write 시스템 콜의 수행시간이 가장 짧은 것으로 측정되었다.

표 8은 위와 동일한 환경에서 원래의 리눅스 커널을 사용할 때와 본 논문에서 수정한 커널을 사용할 때의 시스템 콜 수행시간을 비교한 것이다. 이를 위해 애플리케이션에서 socket, bind, connect 및 close 시스템 콜을 각각 10,000 회씩 호출한 수행시간을 측정하고, 평균값을 계산하였다. 표 8은 이 과정을 5회 측정한 결과이다. 결과를 비교해 보면, 평균적으로 socket 및 close 시스템 콜은 각각 53 ns와 33 ns, bind 및 connect 시스템 콜은 9 ns와 21 ns 정도의 오버헤드를 나타내고 있다. 소켓 시스템 콜은 파일 디스크립터와 같은 파일 시스템의 자료구조를 사용하기 때문에 이에 대한 동기화 및 검색 오버헤드가 존재한다. 그러므로 실제 시스템 콜의 수행의 오버헤드는 표 7보다 큰 값을 보이는 것으로 생각된다. 시스템 콜의 평균 수행시간이 수 μs 인데 비해, 오버헤드는 ns 단위이므로 수정된 시스템 콜은 기존의 소켓에 대해 약간의 오버헤드만을 준다고 볼 수 있다. read 및 write 시스템 콜의 수행시간은 데이타의 로깅시에 파일 입출력의 오버헤드 문제로 정확한 값을 측정하기가 어려웠다. 그러나 read 및 write 시스템 콜은 다른 시스템 콜에 비해서 첨가된 조건문의 수가 적으므로 더 작은 오버헤드를 제공할 것으로 생각된다.

4.2 패킷 처리 시간

embeddedRTP와 ExtendedERTP의 성능 비교를 위

표 7 첨가된 조건문 코드의 수행시간

시스템 콜	socket	bind	connect	read	write	close
추가된 코드의 수행시간 (ns)	4.58	2.27	2.77	1.77	1.81	5.38

표 8 시스템 콜의 평균 수행시간 비교(μsec)

구분	시스템콜	1	2	3	4	5	평균값
원래의 커널	socket	4.2914	4.3016	4.3084	4.1999	4.0967	4.300
	bind	3.3988	3.3961	3.4021	3.3388	3.3339	3.399
	connect	2.292	2.361	2.332	2.3017	2.2881	2.328
	close	2.7908	2.838	2.8067	2.7859	2.7121	2.812
수정된 커널	socket	4.2592	4.281	4.3302	4.3715	4.2946	4.290
	bind	3.3694	3.3726	3.413	3.3675	3.3931	3.385
	connect	2.329	2.3304	2.3471	2.3323	2.3405	2.336
	close	2.7868	2.8313	2.8105	2.8555	2.8166	2.810

해 다음과 같이 측정하였다. 성능 비교를 위해 패킷 처리 시간은 UDP 계층에서 RTP 계층을 통과하는데 걸리는 시간과 애플리케이션이 패킷을 요청하여 가져오기까지의 시간으로 나뉘어 표 5와 같은 PDA 환경에서 측정되었다. 다윈 스트리밍 서버와 MP4LIVE를 이용하여 각각 패킷 처리 시간이 측정된 MP4 파일 스트리밍과 실시간 스트리밍 영상의 프레임과 비트율은 표 9와 같다.

표 9 측정에 사용된 MP4 파일 및 실시간 영상

	해상도	프레임율(fps)	비트율 (kbps)
MP4 파일	192 × 108	11	84
실시간 영상	176 × 144	10	120

성능을 분석하기 위한 샘플 영상을 선택할 때 영상의 프레임율과 비트율이 지나치게 높거나 낮아서 RTP 버퍼 큐에 오버/언더플로우가 발생할 경우, 비동기 모드일 때는 바로 리턴 되고 동기모드일 때는 프로세스가 블록되어 대기하기 때문에 패킷 전송과 추출에 소모되는 처리시간을 제대로 측정할 수 없다. 때문에 성능분석을 위해 MP4LIVE 스트리밍 서버에 대해서는 실시간 스트리밍 영상의 프레임율과 비트율을 적절히 조정하였고, 다윈 스트리밍 서버에 대해서도 적절한 수준의 프레임율과 비트율을 가지는 영상을 선택하여 RTP계층 버퍼에서 패킷 오버/언더플로우가 발생하지 않도록 하였다. 표 10은 패킷 처리 시간을 측정한 결과이다.

표 10에서 볼 수 있듯이 ExtendedERTP의 패킷 처리 시간이 embeddedRTP에 비해 약 15~20% 향상되었음을 확인할 수 있다. 우선, 전송시에 ExtendedERTP는 버퍼 큐에 RTP 패킷의 페이로드를 복사하지 않고, 추가적인 RTP 패킷 헤더의 잔여정보를 생성하지 않는다. 또한, embeddedRTP에서 RTP 계층에 패킷을 저장하기 위해 요구되었던 메모리 청크의 저장위치 탐색 소모시간과 가용메모리 공간을 관리하는 비트맵 갱신 오버헤드[5]가 제거됨으로써 패킷 삽입시 처리시간이 30~40% 가량 향상되었다.

RTP 계층에서 패킷을 추출하는 경우의 처리 시간은 embeddedRTP 보다 지연되었다. 이는 embeddedRTP에서는 애플리케이션에 패킷을 전달하기 위해 RTP 계층에 있는 전체 패킷을 복사해주고 비트맵을 갱신하는 작업만이 요구되었으나, ExtendedERTP에서는 애플리케이션이 패킷을 요청했을 때 UDP 계층의 버퍼에 접근해서 페이로드를 복사하는데 소요되는 지연 때문이다.

4.3 메모리 사용량

embeddedRTP에서는 RTP 세션이 초기화될 때 RTP 계층에 패킷을 저장하기 위한 메모리 풀을 만들고 이를 비트맵을 사용해 유닛단위로 관리하였다. embedded-RTP에서 RTP 세션마다 할당되는 RTP 계층 버퍼 공간의 크기는 최대 크기의 RTP 패킷(1600byte)을 동시에 64개까지 저장할 수 있는 102.4KB(1600 × 64)이다. 하지만, ExtendedERTP는 RTP패킷의 헤더만을 저장하기 때문에 하나의 RTP 패킷을 관리하기 위해 RTP 헤더 구조체 크기(56 byte)의 메모리만이 사용된다. 그러므로, 동일한 n 개의 패킷이 저장될 때, embeddedRTP에 비해 ExtendedERTP를 사용하는 경우 요구되는 버퍼 메모리양은 최소 (56 × n) / (1600 × n) = 3.5%에 수준에 불과하며 RTP 계층에서 저장하는 패킷의 수가 많아질수록 메모리 요구량의 절대적인 수치는 더 차이가 나게 된다.

5. 결론 및 향후 연구 과제

인터넷을 통한 전화, 화상 회의, 원격 교육 등에서 필요로 하는 실시간 데이타 송수신 지원을 위한 프로토콜로서 RTP가 널리 사용되고 있다. 그러나 지금까지 RTP는 라이브러리 계층에서 구현되어 사용됨으로써 다양한 사용자 인터페이스로 인한 혼란을 일으킬 수 있다. 이에 본 논문은 기존의 embeddedRTP를 기반으로 하여 사용자 인터페이스에 대한 편의성을 제공하고 RTP의 성능을 향상시키고자 embeddedRTP의 시스템 콜을 BSD 소켓 API 내로 통합하고 버퍼 메커니즘의 문제점 등을 수정하여 성능을 개선하였다.

본 논문의 ExtendedERTP는 기존 BSD 소켓 API를 통한 시스템 콜과 RTP 지원을 위해 통합된 BSD 소켓 API를 통한 시스템 콜 수행시간을 비교해 보면, 평균 시스템 콜 수행시간이 수 μs 인데 비해 ns 수준의 상대적으로 작은 오버헤드를 보였다. 성능에서는 PDA 환경에서 embeddedRTP에 비해 15~20% 정도 향상된 패킷 처리 속도를 나타내었다. 또한, 동일한 개수의 패킷이 저장될 때 요구되는 버퍼 메모리양은 embeddedRTP

표 10 패킷 처리 시간 비교

영상	기법	삽입시(ms)	추출시(ms)	전체 패킷 처리 시간(ms)
MP4 파일	embeddedRTP	25869.156	7161.44	33030.596
	ExtendedERTP	17714.404	10154.052	27868.456
실시간 영상	embeddedRTP	33888.39	10398.410	44286.8
	ExtendedERTP	22673.92	13127.092	35801.012

에 대비 최소 3.5%에 수준에 불과하였다.

향후에 연구해야 할 과제는 다음과 같다. 우선, RTSP와 같은 RTP의 상위 프로토콜의 커널 레벨에서의 구현이다. 현재의 구조는 RTP만을 커널에 포함시키고 RTSP와 같은 상위 프로토콜은 사용자 영역에서 따로 구현해야 하는 구조이다. 이 구조에서 RTP의 상위 프로토콜을 커널 레벨에 추가적으로 포함시킴으로써 커널 수준에서 세션을 설정 및 해제를 수행하고, 데이타를 전송하는 것이 가능하도록 BSD 소켓 API를 확장하는 연구가 진행된다면 더 좋은 결과를 얻을 수 있을 것이다. 또한 커널 영역에서 UDP 버퍼 큐를 사용하지 않고 바로 RTP 버퍼 큐로 데이타를 수신하는 등의 RTP 자체의 성능을 향상시킬 수 있는 연구가 추가적으로 이루어질 필요가 있다.

참 고 문 헌

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

[2] J. Rosenberg and H. Schulzrinne, "Timer Reconsideration for Enhanced RTP Scalability," INFOCOM '98, 17th Annual Joint Conference of the IEEE Computer and Communications Societies, Proceedings IEEE, pp. 233-241, vol. 1, 1998.

[3] P. Sharma, D. Estrin, S. Floyd and V. Jacobson, "Scalable Timers for Soft State Protocols," INFOCOM '97, 16th Annual Joint Conference of the IEEE Computer and Communications Societies, Proceedings IEEE, pp. 222-229, vol. 1, 1997.

[4] 남성준, 이병래, 김태우, 김태윤, "실시간 멀티미디어 데이타 전송을 위한 SRTPIO 모듈 설계 및 구현," 한국정보과학회 논문지, 제28권, 제4호, pp. 621-630, 2001.

[5] 선동국, 김태웅, 김성조, "내장형 시스템의 원활한 멀티미디어 서비스 지원을 위한 커널 수준의 RTP", 한국정보과학회 논문지, 제10권, 제6호, pp. 460-471, 2004.

[6] H. Schulzrinne, A. Rao, and R. Lanphier, "Real Time Streaming Protocol(RTSP)," RFC 2326, Apr. 1998.

[7] ITU-T Recommendation H.323: "Packet based multimedia communications systems" Feb. 1998.

[8] J. Rosenberg, H. Schulzrinne, G. Carmarillo, A. Johnston R. Sparks, M. Handley, and E. Schooler, "SIP : Session Initiation Protocol," RFC 3261, 2002.

[9] University College London, "UCL Common Multimedia Library," http://www-mice.cs.ucl.ac.uk/multimedia/software/common/index.html.

[10] oRTP, "oRTP - a Real-time Transport Protocol Stack under LGPL," http://www.linphone.org/ortp/.

[11] Lucent Technologies, "Bell Labs, RTPlib," http://www-out.bell-labs.com/project/RTPlib/.

[12] A. Meixner, P. Yin, D. Onyango, and A. Vahdat, "Design and Evaluation of a Kernel-Level SCTP Implementation," Submitted for publication. May 2001.

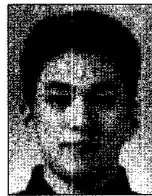
[13] M. Molteni and M. Villari, "Using SCTP with Partial Reliability for MPEG-4 Multimedia Streaming," Proc. of BSDCon Europe 2002, Oct. 2002.

[14] J. Erkkila, "Real-Time Audio Servers on BSD Unix Derivatives," Master's Thesis in Information Technology, Univ. of Jyvaskyla, Filand, June 2005.

[15] Linux DVB API Version 4 (www.linuxtv.org), CE Linux Forum, 2005.

[16] MPEG4IP, "MPEG4IP - Open Streaming Video and Audio," http://www.mpeg4ip.net.

[17] Apple, "Apple - Public Source - Darwin Streaming Server," http:// developer.apple.com/darwin/projects/streaming/.



최 문 선

2003년 중앙대학교 컴퓨터공학과(공학사) 2003년~현재 중앙대학교 컴퓨터공학과 석사과정. 관심분야는 이동컴퓨팅, 임베디드 시스템, RTOS

김 경 산

2004년 중앙대학교 전기전자공학부(공학사). 2004년~현재 중앙대학교 컴퓨터공학과 석사과정. 관심분야는 이동컴퓨팅, 임베디드 시스템, RTOS

김 성 조

1975년 서울대학교 응용수학과(공학사) 1977년 한국과학기술원 전산과(이학석사) 1977년~1980년 ADD(연구원). 1980년~현재 중앙대학교 컴퓨터공학부(교수). 1987년 Univ. of Texas at Austin(공학박사) 1887년~1988년 Univ. of Texas at Austin(Research Fellow). 1996년~1997년 Univ. of California-Irvine(Visiting Professor). 관심분야는 이동컴퓨팅, 임베디드 소프트웨어, 유비쿼터스컴퓨팅