DOI: 10.3745/KIPSTA.2009.16-A.3.189

종단간 QoS 지원을 위해 Bottom-Half 메커니즘을 이용한 우선순위 및 예산 기반의 네트워크 프로토콜 처리

김 지 민[†] · 유 민 수[††]

요 약

종단 호스트에서의 전통적인 네트워크 프로토콜 처리 기법은 인터럽트 기반의 선착순 처리 방식을 사용함으로써 다음과 같은 두 가지 문제점을 가진다. 첫째, 인터럽트가 가장 높은 우선순위로 처리되기 때문에 네트워크 패킷과 응용 프로세스간에 우선순위 역전현상이 발생할 수 있다. 둘째, 네트워크 패킷 처리가 선착순으로 진행되기 때문에 패킷과 패킷간에 우선순위 역전현상이 발생할 수 있다. 이러한 문제는 우선순위에 기반한 프로토콜 처리 방법으로 해결이 가능한 것으로 알려져 있지만, 기본적으로 우선순위 기반의 해법은 기아(starvation)라는 부작용을 가지고 있으며 각각의 네트워크 흐름에 대하여 QoS 격리 및 조절이 불가능하기 때문에 네트워크 흐름마다 상이한 QoS가 요구되는 환경에 적용하기 어렵다는 문제점을 가지고 있다. 본 논문에서는 우선순위와 예산(budget) 개념에 기반하여 bottom-half 메커니즘을 이용한 프로토콜 처리 기법을 제안한다. 제안하는 방법을 사용하면 우선순위 기반의 프로토콜 처리 방법이 가지고 있는 기아 현상을 해결함은 물론 각각의 네트워크 연결이 요구하는 QoS의 격리(isolation)가 가능하다. 이러한 특성으로 인해 패킷 처리 시간을 상한 (upper-bound)시키는 것이 가능해지며, 본 논문에서는 그 최대값을 계산해내는 방법을 함께 제안한다. 마지막으로, 실험을 통해 제안하는 방법이 네트워크 흐름간 QoS를 효과적으로 격리 및 조절할 수 있음을 확인할 수 있었다.

키워드 : Quality of Service(QoS), 네트워크 프로토콜 처리, 우선순위, 운영체제, Bottom-Half

Priority- and Budget-Based Protocol Processing Using The Bottom-Half Mechanism for End-to-End QoS Support

Jimin Kim[†] · Minsoo Ryu[††]

ABSTRACT

The traditional interrupt-based protocol processing at end hosts has two priority-inversion problems. First, low-priority packets may interrupt and delay high-priority process executionssince interrupts have the highest priority in most operating systems. Second, low-priority packet may delay high priority packets when they arrive almost simultaneously since interrupt processing is performed in a FCFS (first come, first served) order. These problems can be solved by a priority-based protocol processing policy and implementation. However, general priority-based schemes commonly have the problem of starvation and cannot support the each network flow requiring the mutually exclusive QoS since the packets are processed in the FCFS order. Therefore, the priority-based schemes are not appropriate for different QoS-demanding applications. In this paper, we present a bottom-half-based approach that relies on priority- and budget-based processing. The proposed approach allows us to solve both the starvation and priority-inversion problems, and further enables effective QoS isolation between different network connections. This feature also enables bounding the protocol processing time at an end host. We finally show through experiments that the proposed approach achieves QoS isolation and control.

Keywords : QoS, Network Protocol Processing, Priority, Operating System, Bottom-Half

1. 서 론

QoS 보장은 네트워크 분야의 중요한 문제의 하나로서 그 동안 네트워크 프로토콜의 각 계층 별로 흥미롭게 다루어져 온 연구 분야이다. 예를 들어, 트랜스포트 계층(transport layer)에서는 MPAT[19], pTCP[20], TCP Nice[23], RSVP[22] 같은 TCP 흐름간의 차별화된 전송성능을 제공하기 위한 연구가 있었으며 네트워크 계층에서도 Integrated Services[5]와 Differentiated Services[6] 같은 차별화된 서비스를 지원하기 위한 연구가 있었다. 링크계층(link layer)과 물리계층(physical layer)에서는 통신 성공률을 높이고 전력소모와 간섭현상을 줄이기 위

※ 이 논문은 2005년 정부(교육인적자원부)의 재원으로 한국학술진흥재단의 지원을 받아 수행된 연구임(KRF-2005-041-D00625).
† 준 회 원 : 한양대학교 전자컴퓨터통신공학과 박사과정
†† 정 회 원 : 한양대학교 컴퓨터공학부 조교수
논문접수 : 2009년 1월 6일
수 정 일 : 1차 2009년 3월 10일, 2차 2009년 3월 31일
심사완료 : 2009년 4월 1일

해 전력제어기법, 복호화기법 등이 연구되었으며 무선환경에서도 QoS 제공을 위한 에러복구 및 무선 네트워크 스케쥴링과 같은 연구가 수행된 바 있다[21]. 그러나 그 동안 QoS와 관련하여 많은 연구에도 불구하고 과거 연구에서는 대부분 네트워크 관점에서의 QoS만을 고려하였던 반면 종단 호스트의 프로토콜 처리기법에 대한 연구는 상대적으로 주목 받지 못하였다.

종단 호스트에서의 프로토콜 처리는 전통적인 인터럽트 기반 선착순 처리 방식을 주로 사용함에 따라 다량의 패킷이 호스트에 도착하여 프로토콜 처리에 경쟁이 발생하는 경우 다음 두 가지 문제점을 가지고 있다. 첫째, 인터럽트는 항상 가장 높은 우선순위로 처리되기 때문에 낮은 우선순위 패킷 처리가 높은 우선순위 어플리케이션의 실행을 예측할 수 없는 시간 동안 지연(delay)시킬 수 있다[7]. 둘째, 네트워크 패킷의 처리가 선착순(first come, first served)으로 진행되기 때문에 패킷 처리간 우선순위 역전현상이 발생할 수 있다.

위와 같은 우선순위역전 문제를 해결하기 위해 기존 연구에서는 우선순위기반의 프로토콜 처리 기법을 제안한 바 있다. 우선순위기반의 프로토콜 처리는 패킷 처리에 우선순위를 고려함으로써 우선순위 역전문제를 해결하는 방식이다. 기존 종단 호스트에서의 우선순위 기반 프로토콜 처리기법은 크게 커널레벨(kernel-level) 프로토콜 처리와 유저레벨(user-level) 프로토콜 처리로 나눌 수 있으며[11-13] 이중에서도 유저레벨 프로토콜 처리기법을 이용한 QoS 제공기법이 주로 연구되었다. 대표적인 커널레벨 프로토콜 처리기법으로써 RIO[2], LRP(Lazy Receiver Processing)[9, 16] 등이 제안된 바 있다. RIO는 'kthread'라고 불리는 운영체제 내부의 커널 쓰레드를 각각의 네트워크 연결에 할당하고 이를 사용하여 프로토콜 처리를 수행하는 방식이다. 또한, kthread는 유저레벨 프로세스와 함께 우선순위에 기반하여 스케쥴링 됨으로써 프로토콜 처리에 우선순위를 반영할 수 있다. LRP 및 [16]에서 제안하는 기법은 어플리케이션이 패킷 수신을 요청할 때까지 커널의 프로토콜 처리를 지연시킴으로써 어플리케이션의 우선순위에 기반하여 프로토콜 처리가 수행될 수 있도록 하였다. 유저레벨 프로토콜 처리기법은 유저공간에서 수행되는 프로세스 또는 쓰레드를 사용하여 프로토콜 처리를 수행하는 기법으로써 [1, 3, 4, 14, 17] 등 다수의 논문에서 소개된 바 있다.

우선순위 기반 네트워크 프로토콜 처리기법이 앞서 언급한 우선순위 역전문제를 효과적으로 해결할 수 있지만 각각의 네트워크 흐름에 대하여 QoS 격리 및 조절이 불가능하기 때문에 낮은 우선순위 프로토콜 처리가 기아상태에 빠질 수 있는 부작용을 가지고 있으며 네트워크 흐름마다 상이한 QoS 요구를 지원하기 어렵다는 문제점을 가지고 있다. 예를 들어 우선순위 기반 네트워크 프로토콜 처리기법에서는 높은 우선순위를 가지는 네트워크 흐름에 과부하(bursty arrival)가 발생할 경우, 상당한 CPU 시간을 요구하는 프로토콜 처리[8]를 높은 우선순위 네트워크 흐름이 독점하여 낮은 우선순위 패킷 처리가 기아(starvation)상태에 빠질 수 있다. 즉, 우선순위기반 프로토콜 처리 방식에서는 낮은 우선순위 네트워크 흐름에 대하여 시스템이 약속한 QoS 제공이 어렵다는 문제점을 가지고 있다. 이를 위해서는 네트워크 흐름당 프로토콜 처리에 사용되는 CPU 자원을 제어할 수 있는 메커니즘이 필요하다.

이와 같은 문제를 해결하기 위해 본 논문에서는 bottom-half 메커니즘을 이용한 우선순위와 예산 기반의 네트워크 프로토콜 처리기법을 제안하며 아울러 최대 패킷처리 시간에 대한 분석방법을 소개한다. 제안하는 기법에서의 우선순위 기반 프로토콜 처리는 개념적으로 기존에 제안된 방식과 동일하나 구현 방법에 있어서 차별화 된다. 기존 연구에서는 유저레벨 쓰레드 또는 커널레벨 쓰레드를 사용하여 프로토콜 처리에 우선순위를 반영하였지만 본 논문에서는 bottom-half 에서 우선순위를 반영할 수 있도록 하였다. 일반적으로 프로세스 또는 쓰레드를 이용한 프로토콜 처리기법은 컨텍스트 전환(context switching), 쓰레드 관련 큐(queue)조작 그리고 스케쥴링 등 쓰레드 관리에 상당한 오버헤드를 요구할 뿐만 아니라 데이터 전달을 위한 IPC(inter-process communications) 비용도 요구한다. 반면에 bottom-half에서는 쓰레드 관리에 수반되는 오버헤드를 요구하지 않으며 기존 운영체제의 네트워크 메시지(message) 전달 기법을 그대로 사용할 수 있다. 제안하는 기법에서는 프로토콜 처리를 수행하는 핸들러(handler)가 bottom-half에서 실행된다. 프로토콜 처리 핸들러는 각각의 네트워크 흐름(flow)에 할당되고 실행 우선순위를 가지며 이에 기반하여 스케쥴링 된다.

제안하는 예산 기반 프로토콜 처리기법은 각 네트워크 흐름당 프로토콜 처리시간을 예산을 통해 제어하는 방식으로써 프로토콜 처리 핸들러는 매 주기마다 해당 네트워크 흐름에 할당된 프로토콜 처리 실행예산(execution-budget)을 충전 받아, 주어진 예산 내에서 패킷을 처리한다. 이를 통해 우선순위 기반 프로토콜 처리방식의 부작용을 최소화할 수 있을 뿐만 아니라 각각의 네트워크 흐름(flow)에 대하여 QoS 격리(isolation) 및 조절을 가능케 할 수 있다. 아울러 본 논문에서는 패킷처리 시간분석을 통하여 제안하는 기법이 최대 패킷처리 시간을 한정(bound) 시킬 수 있음을 설명한다.

본 논문에서는 우선순위와 예산기반의 프로토콜 처리를 통해 기존 프로토콜 처리 기법에서는 제어하기 어려웠던 QoS 격리 및 조절이 가능함을 주장하고 실험을 통해 이를 입증하였다. 제안된 기법은 인터럽트 기반 선착순 프로토콜 처리방식을 사용하는 리눅스 네트워크 서브시스템을 수정하여 구현되었다.

본 논문의 구성은 다음과 같다. 2장에서는 전통적인 인터럽트 기반의 선착순 프로토콜 처리방식의 문제점 그리고 제안하는 기법및 소프트웨어 구조에 대하여 설명한다. 3 장에서는 제안하는 기법이 수신된 패킷의 최대 패킷처리 시간을 한정(bound) 시킬 수 있음을 분석(analysis)을 통해 입증한다. 4장에서는 구현 및 실험에 대하여 설명하고 5장에서 본 논문의 결론을 짓는다.

2. 제안하는 네트워크 프로토콜 처리기법

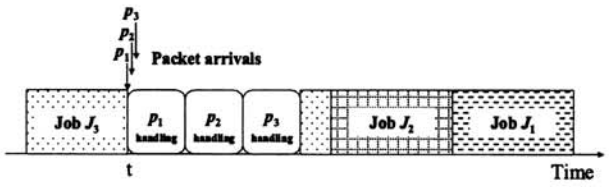
본 장에서는 전통적인 네트워크 프로토콜 처리방식에서 발생할 수 있는 문제점을 소개하고 이를 해결하기 위한 프로토콜 처리기법과 소프트웨어 구조를 제안한다. 다음 <표 1>은 본 논문에서 사용된 기호를 정리한 것이다.

〈표 1〉 논문에서 사용된 기호요약(notation summary)

기호	설 명
J_i	우선순위 i를 갖는 어플리케이션
p_i	우선순위 i를 갖는 패킷 (주의: 소문자 p)
F_i	i번째 네트워크 흐름
P_i	i번째 네트워크 흐름의 우선순위 (주의: 대문자 P)
S_i	i번째 네트워크 흐름의 패킷 대기큐(wait queue)
H_i	i번째 네트워크 흐름의 프로토콜 처리를 전담하는 핸들러
T_i	핸들러 H_i 의 예산 충전주기
B_i	핸들러 H_i 의 실행예산 충전량 (단위: CPU 사용시간)
S	바이트당 프로토콜 처리시간
D_i	토큰 버킷(token bucket) 모델 아래에서 i번째 네트워크 흐름의 깊이(depth)
R_i	토큰 버킷(token bucket) 모델 아래에서 i번째 네트워크 흐름의 패킷 수신율(arrival rate)
W_i	i번째 네트워크 흐름의 프로토콜 처리 대기시간
I_i	i번째 네트워크 흐름의 프로토콜 처리가 자신보다 같거나 높은 우선순위를 가진 프로세스 또는 패킷처리에의해 지연되는 시간
C_i	i번째 네트워크 흐름으로부터 수신된 처리해야 할 바이트 단위의 패킷크기(bytes size)
Q_i	i번째 네트워크 흐름으로부터 수신된 패킷의 프로토콜 처리시간

2.1 인터럽트 기반 선착순 처리방식에서의 우선순위 역전

전통적인 프로토콜 처리기법은 일반적으로 인터럽트 기반의 선착순 방식을 사용함에 따라 '우선순위 역전'을 발생시킬 수 있다[15]. (그림 1)과 같이 어플리케이션 J_3가 실행 중일 때, 패킷 p_1, p_2, p_3가 t 시점에 차례대로 도착한다고 가정하자. 아래 첨자의 숫자가 클수록 높은 우선순위를 나타낸다. 예를 들어 p_2는 p_1보다 높은 우선순위를 가지지만 J_3보다는 낮은 우선순위를 가진다. 첫 번째 패킷 p_1이 t 시점에 도착할 때, 인터럽트는 항상 어플리케이션보다 앞서 처리되기 때문에 전통적인 방식에서는 J_3가 즉시 멈추고 패킷 p_1의 처리가 시작된다. 그리고 p_1의 처리가 끝난 후, 두 번째 패킷 p_2의 처리가 선착순 방식에 의해 곧바로 시작된다. 여기에서 두 가지 종류의 우선순위 역전현상을 관찰할 수 있다. 첫 번째 종류의 우선순위 역전은 패킷과 어플리케이션간에 발생한다. 위의 예제에서 패킷 p_1은 J_3보다 낮은 우선순위를 가지고 있지만 프로세스 J_3를 선점한다. 두 번째 종류의 우선순위 역전현상은 패킷간에 발생한다. 패킷 p_2는 p_3보다 우선순위가 낮지만 선착순 방식에 의해 p_3보다 앞서 처리된다. 이러한 문제는 각각의 네트워크 연결에 우선순위를 할당하고 우선순위에 기반한 프로토콜 처리를 수행함으로써 해결될 수 있다[1, 2, 4, 13, 14].



(그림 1) 우선순위 역전현상

우선순위 기반 네트워크 프로토콜 처리기법이 위와 같은 우선순위 역전문제를 효과적으로 해결할 수 있지만 각각의 네트워크 흐름에 대하여 QoS 격리 및 조절이 불가능하기 때문에 네트워크 흐름마다 상이한 QoS 지원이 어렵다는 문제점을 가지고 있다. 예를 들어 우선순위 기반 네트워크 프로토콜 처리기법에서는 높은 우선순위를 가지는 네트워크 흐름에 과부하(bursty arrival)가 발생할 경우, 높은 우선순위 네트워크 흐름이 프로토콜 처리를 독점하여 낮은 우선순위 패킷 처리가 기아상태에 빠질 수 있다. 즉, 우선순위기반 프로토콜 처리 방식에서는 낮은 우선순위 네트워크 흐름에 대하여 시스템이 약속한 QoS 제공이 어렵다는 문제점을 가지고 있다. 이를 위해서는 네트워크 흐름당 프로토콜 처리에 사용되는 CPU 자원을 제어할 수 있는 메커니즘이 필요하다.

2.2 우선순위 및 예산 기반의 네트워크 프로토콜 처리

위 문제를 해결하기 위해 본 논문에서는 bottom-half 메커니즘을 이용한 우선순위와 예산 기반의 네트워크 프로토콜 처리 기법을 제안한다. 제안하는 기법은 우선순위 기반의 프로토콜 처리의 부작용을 최소화할 수 있으며 프로토콜 처리에 필요한 CPU 자원을 실행예산으로 할당할 수 있어 특정 네트워크 흐름의 프로토콜 처리 독점문제를 해결할 수 있다. 뿐만 아니라 종단 호스트가 수용할 수 없는 과도한 통신이 발생할 경우, 수신 라이브락(receive livelock)[7]과 같은 현상이 발생할 수 있는데 이와 같은 문제에서도 제안하는 기법이 효과적으로 해결할 수 있을 것으로 예상한다. 본 논문에서는 페이지 제한으로 수신 측(receive-side) 프로토콜 처리만을 설명한다. 그러나 제안하는 방법은 송신 측(send-side) 프로토콜 처리에도 적용될 수 있다.

본 절에서는 제안하는 기법의 실행 메커니즘에 대해 설명하도록 한다. 우선순위는 2개 이상의 핸들러가 프로토콜 처리를 위해 CPU자원을 두고 경쟁하는 경우 이를 스케쥴링하기 위해 사용되며 실행예산은 CPU자원의 사용시간을 결정하는데 사용된다. 즉 실행예산은 해당 네트워크 흐름의 프로토콜 처리를 위한 CPU 점유시간으로 사용되며 우선순위는 실행예산이 남아있는 다수의 네트워크 흐름에 패킷이 수신된 경우 패킷처리를 스케쥴링하기 위해 사용된다. 비록 네트워크 흐름의 우선순위가 높다 할지라도 실행예산이 없다면 수

신된 패킷은 다음 예산할당 때까지 대기해야 하며 패킷처리 경쟁(스케줄링)에 참여할 수 없다. 핸들러 스케줄링은 오직 우선순위에 의해서만 결정되고 실행예산은 스케줄링에 영향을 주지 않는다. 구현을 위한 설계는 2.3절에서 다루도록 한다.

우선순위 기반 프로토콜 처리: 우선순위 기반 프로토콜 처리를 위해 제안하는 기법에서는 임의의 네트워크 흐름 F_i를 $<P_i, S_i, H_i>$로 표현한다. P_i는 네트워크 흐름 F_i의 우선순위이며 S_i는 F_i로부터 수신된 패킷이 프로토콜 처리를 기다리기 위한 대기큐(wait queue)이다. 그리고 H_i는 F_i의 프로토콜 처리를 전담하는 핸들러로써 우선순위 P_i 를 상속받는다. 새로운 네트워크 흐름 F_i가 생성되는 경우 대기큐 S_i와 핸들러 H_i도 함께 생성되어 F_i에 할당된다. 네트워크 흐름 F_i에 수신된 패킷은 네트워크 흐름 별로 분류되어 해당 대기큐 S_i에 삽입된다. 해당 핸들러 H_i 는 프로토콜 처리에 사용될 예산이 남아있고 해당 우선순위 P_i 가 다른 핸들러 및 어플리케이션의 우선순위 보다 높을 경우 자신에게 주어진 실행예산을 소비하면서 패킷을 처리한다.

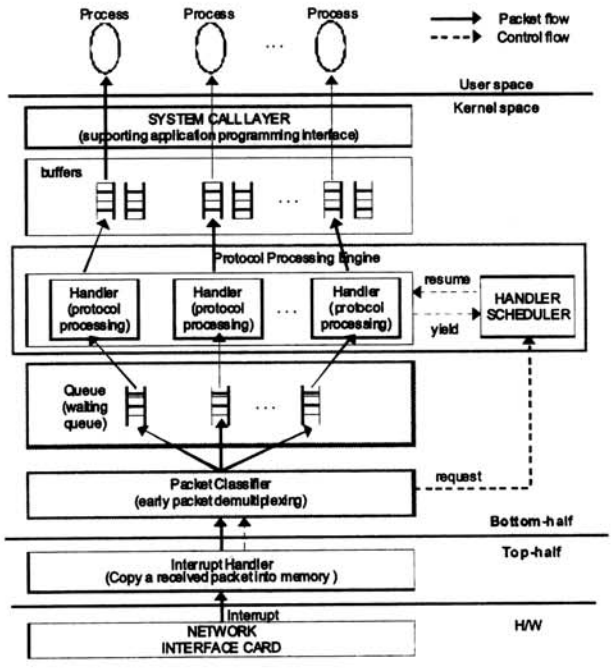
예산 기반 프로토콜 처리: 제안하는 핸들러 H_i는 주기 T_i에서 프로토콜 처리 실행예산 B_i를 할당 받고 우선순위에 의해 실행이 결정된다. 실행예산 B_i의 단위는 프로토콜 처리를 위한 CPU 사용시간이다. 프로토콜 처리 핸들러 H_i는 매 패킷처리 후 프로토콜 처리에 사용된 CPU 시간만큼 자신에게 할당된 실행예산을 감소시킨다. 만약 핸들러 H_i가 프로토콜 처리에 필요한 실행예산을 전부 소비해버린 경우 패킷은 다음 주기에서 새로운 실행예산 B_i가 할당될 때까지 대기큐 Si에서 대기한다. 핸들러 H_i는 자신의 실행예산을 현재 주기 내에서 전부 소비하지 못할 수도 있다. 이러한 경우 소비되지 않은 실행예산은 버려지게 되고 다음 주기에서 새로운 실행예산 Bi를 할당 받는다. 이러한 방법으로 특정 네트워크 흐름의 프로토콜 처리율을 최대 B_i/T_i 로 제한할 수 있다. 아울러 B_i/T_i은 해당 네트워크 흐름F_i의 프로토콜 처리를 위한 CPU 이용률(utilization proportion)로 간주될 수 있다.

2.3 제안하는 기법의 소프트웨어 구조

제안하는 기법에서는 bottom-half 핸들러가 각각의 네트워크 흐름에 할당되고 해당핸들러에 의해 프로토콜 처리가 수행된다. 기존에 제안된 우선순위기반 처리 기법에서는 우선순위 비교 메커니즘을 프로토콜 처리에 적용하기 쉽도록 주로 쓰레드를 사용하여 프로토콜 처리를 수행하였다. 쓰레드 방식은 기존 CPU 스케줄러의 스케줄링 정책을 사용할 수 있기 때문에 구현이 비교적 용이하다는 장점을 가지고 있다. 그러나 외부 이벤트에 대한 응답성과 오버헤드는 일반적으로 bottom-half 메커니즘이 쓰레드 방식에 비해 유리하기 때문에 본 논문에서는 bottom-half 메커니즘을 이용한 우선순위 및 예산기반의 프로토콜 처리 기법을 제안한다. 제안하는 기법의 단점으로써 각 네트워크 흐름에 할당된 프로토콜 처리 핸들러는 커널에 내장된 CPU 스케줄러의 스케줄링 대상이 아니기 때문에 핸들러를 우선순위에 기반하여 스케줄링할 수 있는 별도의 핸들러 스케줄러가 필요하다. 또한 쓰레드 방식에서는 프로토콜 처리 중 CPU 강제 선점이 가능하지만 제안하는 기법에서는 핸들러가 bottom-half 컨텍스트에서 수행되기 때문에 실행 중 강제 선점이 불가능하다. 따라서 핸들러는 더 높은 우선순위 실행객체가 도착한 경우 CPU를 스스로 양도해야만 한다.

(그림 3)은 제안하는 기법의 소프트웨어 구조를 나타낸 것으로써 패킷 분류기, 대기큐, 프로토콜 처리 핸들러 그리고 핸들러 스케줄러로 구성된다. 패킷 분류기는 네트워크 서브시스템에 패킷이 도착한 경우 패킷이 해당 대기큐에 삽입될 수 있도록 패킷 역 다중화(early packet demultiplexing) 기능을 수행한다[9]. 핸들러 스케줄러는 핸들러 및 어플리케이션의 우선순위를 비교하여 핸들러의 실행 유무를 결정하는 기능을 담당한다.

제안하는 기법에서는 네트워크를 통해 패킷이 수신된 경우, 커널은 현재 수행중인 어플리케이션을 중단 시키고 top-half의 네트워크 인터럽트 핸들러를 실행시킨다. 인터럽트 핸들러는 bottom-half 핸들러에게 프로토콜 처리를 요청하면서 top-half 작업을 종료한다. 이 후 Bottom-half에서는 수신된 패킷 데이터를 시스템 메모리로 복사하고 패킷 분류기를 호출한다. 호출된 패킷 분류기에서는 패킷을 해당 대기큐에 삽입하고 핸들러 스케줄러를 호출한다. 핸들러 스케줄러는 실행할 핸들러를 우선순위에 기반하여 선택하고 선택된 핸들러에게 프로토콜 처리를 위한 CPU 점유권을 넘긴다. CPU 점유권을 받은 핸들러는 자신에게 할당된 실행예산을



(그림 3) 제안하는 네트워크 서브시스템의 구조

소비하면서 해당 대기큐의 패킷을 선착순으로 처리한다. 주의할 점은 Bottom-half에서는 핸들러 스케줄러가 핸들러를 강제 선점할 수 없기 때문에, 우선순위 기반 패킷처리를 위해서 핸들러는 매 패킷 처리 후 자신 보다 높은 우선순위 패킷이 프로토콜 처리 중에 도착하였는지를 확인해야 하며 만약 자신보다 높은 우선순위 패킷이 도착하였다면 CPU 제어권을 다시 핸들러 스케줄러에게 양도해야만 한다. 프로토콜 처리 중 새로도착한 패킷에 대한확인 및 처리는 NAPI (New Api)의 폴링(polling)방식을 이용함으로써 수행된다. 즉, 프로토콜 처리 중 새로 도착한 수신 패킷은 프로토콜 처리 컨텍스트에서 NIC(network interface card)를 재검사함으로써 인터럽트 없이 패킷의 수신유무를 확인할 수 있다. 이때 수신된 패킷은 커널 메모리로 복사되고 패킷 분류기가 재호출되어 수신된 패킷을 해당 대기큐에 삽입한다. 따라서 제안하는 기법은 NAPI의 폴링 메커니즘의 사용을 가정하고 있으며 현재 대부분의 NIC 드라이버들이 NAPI의 폴링 메커니즘을 지원하고 있다.

예산기반의 프로토콜 처리를 위해 핸들러는 실행예산이 부족한 경우 대기상태가 되고 제어권을 다시 핸들러 스케줄러에게 양도한다. 이때 핸들러의 실행예산 충전은 타이머 핸들러에 의해 주기적으로 수행될 수 있다. 따라서 타이머는 예산충전 주기가 경과되었는지를 알아보기 위해 모든 핸들러를 반복적으로 검사해야 한다. 그러나 핸들러의 실행예산은 패킷이 존재하는 경우에만 의미가 있기 때문에 이와 같은 오버헤드를 줄이기 위해 본 논문에서는 패킷이 수신된 경우 패킷 분류기가 직접 타이머 값을 참조하여 해당 핸들러의 주기경과 유무를 판단하고 주기가 경과된 경우 실행예산을 충전해 준다. 또한 패킷 분류기는 대기상태의 핸들러가 처리해야 할 패킷이 있고 프로토콜 처리에 소비할 예산이 있는 경우 핸들러의 상태를 대기상태에서 준비상태로 만들어 준다. 반대로 패킷이 없거나 예산을 전부 소비한 경우 핸들러는 준비상태에서 대기상태로 전환되고 스케줄링 경쟁에 참여할 수 없다. 즉, 핸들러 스케줄러는 오직 준비상태인 핸들러만을 스케줄링 대상으로 고려한다.

제안하는 기법으로 인한 주요 추가 오버헤드로는 반복적인 폴링 함수호출, 패킷분류기 실행 그리고 핸들러 스케줄링 오버헤드가 있다. 폴링 함수에서는 패킷 수신유무를 확인하기 위해 NIC카드의 상태 레지스터를 검사하고 패킷이 수신되지 않은 경우 폴링 함수는 수행을 종료하고 호출함수(caller)로 복귀한다. 패킷 분류기는 패킷 헤더(header)의 포트(port) 번호를 식별하여 해당 대기큐에 삽입한 후 해당 핸들러의 예산충전 주기가 경과되었는지를 확인하는 간단한 작업을 수행한다. 핸들러 스케줄러 역시 핸들러 우선순위와 현재 수행중인 어플리케이션의 우선순위를 고려하여 핸들러의 실행유무를 결정하는 매우 간단한 작업을 수행한다. 이와 같은 함수호출 오버헤드, 상태 레지스터 검사, 포트번호 식별 그리고 우선순위 비교 오버헤드는 상당한 CPU 시간을 요구하는 프로토콜 처리 오버헤드에[8] 비해 매우 작기 때문에 제안하는 구조로 인한추가 오버헤드가 기존 네트워크 처리성능에 미치는 영향은 매우 작다고 할 수 있다.

제안하는 소프트웨어 구조를 송신측에 적용하기 위해서는 다음 사항을 고려해야 한다. 일반적으로 패킷 송신은 어플리케이션이 송신 데이타를 생성하고 시스템 콜을 통해 해당 송신버퍼(outgoing buffer)에 데이터를 삽입한 후 프로토콜 처리가 수행된다. 그러나 송신버퍼가 각각의 네트워크 흐름당 할당되어 있지 않고 전역적으로 사용된다면 패킷 스케줄링을 위해 네트워크 흐름당 송신버퍼를 생성해주고 송신 시스템 콜에서 패킷 분류기를 호출하여 송신 데이터를 해당 송신버퍼에 삽입해 주어야 한다. 또한 핸들러 스케줄러는 커널 내부 모듈로써 구현되어야 하며 패킷 처리를 스케줄링 할 수 있어야 한다. 송신 시스템 콜인 경우 프로토콜 처리 핸들러는 bottom-half 컨텍스트가 아닌 시스템 콜 컨텍스트에서 동작하며 송신 시스템 콜의 나머지 작업을 수행한다. 기타 각 모듈의 동작 메커니즘은 수신측과 유사하다.

3. 최대 패킷처리 시간분석(protocol processing latency)

일반적으로 시간제약이 엄격한 시스템에서는 시스템의 예측성(predictability) 및 결정성(determinacy)이 성능에 매우 중요한 고려 요소가 된다. 본 장에서는 호스트의 작업들(jobs)이 스케줄링 가능한 상황에서, 제안하는 기법이 수신된 패킷을 예측된 시간 안에 처리할 수 있음을 설명한다. 이것은 제안하는 기법이 엄격한 시간제약 시스템에서도 효과적으로 사용될 수 있음을 나타낸다.

제안하는 기법에서 프로토콜 처리율 B_i/T_i가 CPU 이용률(utilization proportion)로 간주될 수 있기 때문에 실시간 CPU 스케줄링 이론을 적용한 분석이 가능하다. Liu와 Layland는 RM(rate monotonic)에서 전체 CPU 이용률이 $ln2$ ($\approx$0.6931)를 초과하지 않는다면 모든 작업(job)은 자신의 주기 안에서 처리될 수 있음을 보였으며 EDF(earliest deadline first)에서는 전체 CPU 이용률이 1을 넘지 않는다면 모든 작업(job)은 스케줄링 가능하다는 것을 보여 주었다[10]. 따라서 프로토콜 처리시간을 포함한 종단 호스트의 전체 CPU 이용률이 해당 스케줄링 정책의 schedulable utilization bound보다 크지 않다면 모든 작업은 시간제약을 만족시킬 수 있으며 이와 동시에 모든 핸들러의 실행시간 B_i 는 보장받을 수 있다. 예를 들어 종단 호스트가 RM 모델을 따르고 전체 CPU 이용률이 $ln2$를 초과하지 않는다면 핸들러는 매 주기 T_i에서 프로토콜 처리 실행예산 B_i를 항상 보장받을 수 있다.

본 논문에서의 패킷처리 시간은 종단 호스트에 패킷이 도착한 시점부터 패킷처리가 완료되는 시점까지의 시간 간격으로 정의한다. 네트워크로부터 수신된 패킷은 자신보다 높거나 같은 우선순위 패킷처리와 프로세스 실행 그리고 핸들러의 프로토콜 처리 예산 부족으로 인하여 프로토콜 처리가 지연(delay)될 수 있다. 일반적으로 우선순위 기반의 스케줄링 정책 아래에서 네트워크 흐름 F_i로부터 수신된 패킷의

프로토콜 처리 시간은 다음 수식으로 표현될 수 있다.

$$Q_i = W_i + I_i + C_i\delta \quad (1)$$

W_i는 프로토콜 처리를 위한 충분한 실행예산이 충전될 때까지 패킷이 대기해야 하는 시간이다. 그리고 I_i는 자신보다 같거나 높은 우선순위를 가진 프로세스 또는 패킷처리에 의해 지연되는 시간으로써 정확한 I_i는 CPU 스케줄링에 의존한다. C_i는 처리해야 할 바이트 단위의 패킷크기(bytes size), δ는 바이트당(per-byte) 프로토콜 처리시간이다. 이 때 최대 패킷처리 시간 Q_i^{MAX}는 전체 CPU 이용률(utilization)이 스케줄링 가능한 CPU 이용률을 초과하지 않으며 네트워크 흐름이 깊이(depth)가 D_i이고 수신율(arrival rate) R_i인 토큰 버킷(token bucket)모델을 따른다고 가정함으로써 한정(bound)될 수 있다. 따라서 모든 프로세스가 스케줄링 가능한 상황에서 각각의 프로토콜 처리 핸들러 H_i는 매 주기 T_i 마다 자신의 실행예산 B_i의 소비를 보장받는다. (그림 2)는 패킷처리 시간 Q_i가 최대가 되는 상황을 보여주고 있다. 이때 $kT_i + B_i$ 시점에 처리되어야 할 패킷들의 크기 총합은 b_i *byte*라고 가정한다. 그리고 기호 $[x]$는 실수(real number) x보다 크거나 같은 수 중에서 가장 작은 정수로 정의한다. 여기서 k와 n은 정수이다.

(그림 2)에서 kT_i+B_i 시점에 도착한 패킷은 다음예산 충전 주기까지 기다린 후 예산을 할당 받아 처리될 수 있다. 그러나 한번의 예산충전 B_i로는 kT_i+B_i 시점에 도착한 모든 패킷들을 처리하지 못할 수도 있기 때문에 프로토콜 처리에 1번 이상의 예산충전 주기가 필요할 수도 있다. 즉, 패킷이 도착한 시점부터 $n(=b_i\delta/B_i-1)$번의 주기가 경과되고 나서야 프로토콜 처리를 위한 예산할당 누적총합이 $bi\delta$ 보다 커질 수 있다. 따라서 W_i^{MAX}는 $(T_i-B_i)+(b_i\delta/B_i-1)T_i$가 된다. I_i는 최악의 스케줄링 상황에서 최대 $(T_i - B_i)$까지 커질 수 있으며 kT_i+Bi 시점에 도착한 패킷들은 최대 $(k+n)T_i$ 시점에서 패킷처리가 완료될 수 있기 때문에 $I_i^{MAX}+a^{MAX}$는 T_i가 된다. 따라서 수식 (1)은 다음과 같이 표현될 수 있다.

$$\begin{aligned} Q_i^{MAX} &= W_i^{MAX} + I_i^{MAX} + a^{MAX} = W_i^{MAX} + T_i \\ &= \{(T_i - B_i) + (\left\lceil \frac{b_i\delta}{B_i} \right\rceil - 1)T_i\} + T_i \\ &= (1 + \left\lceil \frac{b_i\delta}{B_i} \right\rceil)T_i - B_i \end{aligned} \quad (2)$$

수식(2)는 네트워크 흐름 F_i의 종단간 최대 패킷처리 시간 Q_i^{ete}를 얻는데 사용될 수 있다. 예를 들어 네트워크에서의 최대 패킷전송 시간을 Q_i^{net}이라 한다면 Q^{ete}는 다음과 같은 수식으로 표현할 수 있다.

$$Q_i^{ete} = Q_i^{net} + Q_{i,s}^{MAX} + Q_{i,r}^{MAX} \quad (3)$$

$Q_{i,s}^{MAX}$와 $Q_{i,r}^{MAX}$는 각각 종단 호스트의 송신 측과 수신 측의 최대 패킷처리 시간을 의미한다. Q_i^{net}은 Integrated Services[5], Differentiated Services[6]와 같은 기법에서 소개된 정량적 분석 모델로부터 구해질 수 있다.

4. 프로토타입 구현 및 실험

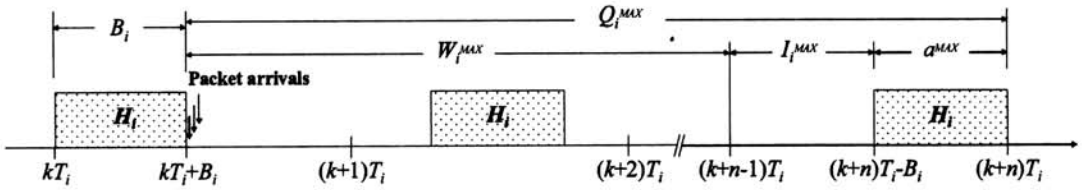
본 논문에서는 제안하는 아키텍처를 리눅스 커널 2.6버전의 네트워크 서브시스템에 적용하여 구현하였다. 리눅스 커널은 네트워크 카드에 수신된 패킷을 top-half와 bottom-half로 나누어 처리하며 전통적인 인터럽트 기반의 선착순 방식으로 패킷을 처리한다. 리눅스 커널버전 2.6에서 지원하는 bottom-half 메커니즘은 softIRQ, 태스크릿(tasklet), 워크큐(work queue)가 있다[18]. 이 중 본 논문에서는 softIRQ 메커니즘을 사용하여 구현하였다.

본 논문에서는 리눅스 네트워크 서브시스템에 제안하는 아키텍처를 구현하기 위해 패킷 분류기와 핸들러 스케줄러 그리고 프로토콜 처리 핸들러를 커널 함수로 구현하였다.

<표 2>은 리눅스 커널에 추가한 시스템 콜(system call)

〈표 2〉 제안하는 네트워크 서브시스템을 구현하기 위해 추가된 주요 함수

함수 원형	기 능
int QoS_channel_setup(int period, int budget)	QoS 채널 생성
void packet_classifier(void)	패킷 분류
void wait_queue_create(int channel_ID)	대기큐(wait queue) 생성
void handler_create(int channel_ID)	핸들러 생성
void handler_scheduler(void)	핸들러 스케줄링
void do_recv_protocol_processing(int channel_ID)	대기큐에 있는 패킷처리
void handler_destroy(int channel_ID)	핸들러 해제
void wait_queue_destroy(int channel_ID)	대기큐 해제
void QoS_channel_close(int channel_ID)	QoS 채널 해제
void cpu_yield(void)	CPU 양도



(그림 2) 패킷처리 시간이 최대가 되는 핸들러 스케줄링 상황

및 주요 함수를 나타낸 것이다.

실험에서는 Pentium4 2.8GHz, Realtek 8139D NIC 그리고 512MB RAM을 사용하였으며 3개의 TCP/IP네트워크 세션을 생성하여 각각 1,000 개의 패킷을 수신하는데 소요되는 시간을 측정하였다. 3개의 네트워크 세션으로부터 패킷을 수신 받는 각각의 네트워크 어플리케이션은 서로 다른 우선순위를 가지도록 하였으며 패킷당 300μs의 CPU 시간을 응용계층에서 소비하도록 하였다. 아울러 프로토콜 처리에 사용된 모든 핸들러의 실행예산 충전 주기는 1초로 설정하였다.

QoS는 end-to-end latency, 초당 프레임수(frames/s), packet loss rate, delay jitter, 패킷 처리율(throughput) 등 다양한 파라미터로 나타낼 수 있다. 본 논문에서는 패킷 처리율을 QoS 측정 파라미터로 사용하였다. (그림 4, 5, 6) 그래프의 X 축은 수신된 패킷의 개수를 나타내며 Y축은 경과된 시간을 나타낸다. 이 때 패킷 처리율은 각 네트워크 흐름별 그래프의 X/Y 기울기로 확인할 수 있으며 그래프가 X축에 가까울수록 높은 프로토콜 처리율을 의미한다. 예를 들어 (그림 4)에서 패킷 수신구간 0~1,000개의 F_1의 패킷 처리율은 1,000packets/144s이고 F_3의 패킷 처리율은 1,000packets/100s이다. 따라서 F_3이 F_1보다 더 높은 프로토콜 처리율을 보이고 있음을 알 수 있다.

(그림 4)와 (그림 5)의 비교 결과, 각 네트워크 흐름이 1,000개의 패킷을 수신하는 데 걸리는 시간 그리고 F_1, F_2, F_3 의 패킷 처리율이 기존 인터럽트 기반의 선착순 방식을 사용하는 리눅스에서도 어느 정도 우선순위에 의존하는 결과를 보였지만 제안하는 방법에서 더욱 우선순위에 의존하는 결과를 보이고 있다. (그림 5)에서 가장 낮은 우선순위 네트워크 흐름 F_1이 대략 200개의 패킷을 수신하고 나서는 거의 일정한 기울기를 보이고 있는데 이것은 다른 네트워크 흐름과 경쟁이 없어지는 지점이기 때문이다. 따라서 이 시점부터는 실행예산 할당량에 내에서 최고 속도로 프로토콜 처리를 수행하고 있음을 알 수 있다. 마찬가지로 (그림 5)에서 네트워크 흐름 F_2가 대략 400개의 패킷을 수신하고 난 후부터는 가장 높은 우선순위가 되기 때문에 이 후부터는 거의 일정한 기울기를 보이고 있다.

(그림 6)은 (그림 5)와 동일한 조건에서 각각의 네트워크 흐름에 할당된 예산량을 변경한 후 측정한 결과이다. 가장 낮은 우선순위 네트워크 흐름 F_1에는 예산량을 300μs에서 200μs로 낮추었고 중간 우선순위 네트워크 흐름 F_2에는 예산량을 300μs로 이전과 동일하게 할당하였으며, 가장 높은 우선순위 네트워크 흐름 F_3에는 300μs에서 400μs로 예산량을 높였다. <표 3>은 예산량 변화를 나타낸 것이다. 본 실험은 예산량 변화에 따른 패킷 처리율의 변화를 알아보기 위한 실험이다.

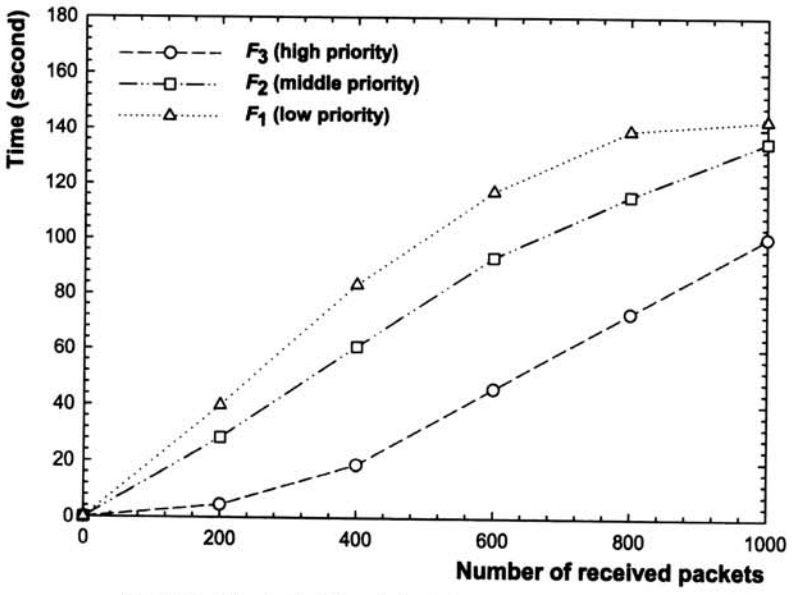
(그림 6)을 (그림 5)와 비교해 보면 (그림 6)의 F_1의 프로토콜 처리율은 (그림 5) 에서 보다 떨어졌으며 F_2의 처리율은 이전과 거의 동일한 상태를 유지했고 F_3는 이전 실험결과 보다 높아진 프로토콜 처리율을 보였다.

(그림 7)은 하나의 네트워크 세션만을 생성한 후, 예산량 변화에 따른 패킷 처리율을 측정한 결과이다. 그래프는 예상한 바와 같이 프로토콜 처리실행 예산량이 많을수록 높은 패킷 처리율을 보이고 있다.

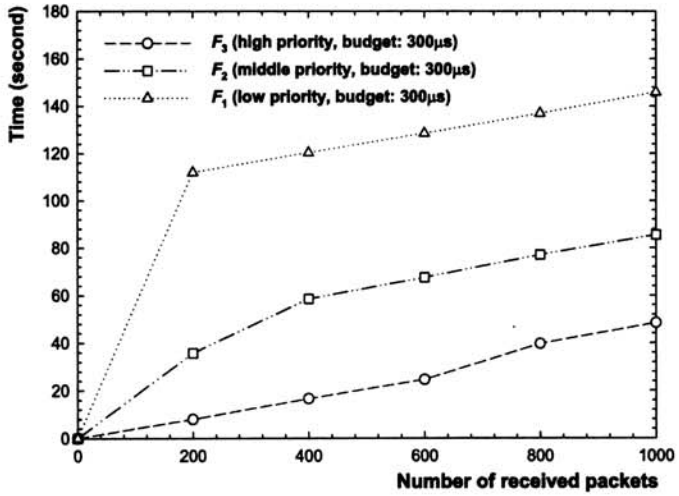
(그림 4, 5, 6, 7)의 실험결과를 통해 제안하는 기법이 우선순위와 실행 예산량을 조절함으로써 기존 네트워크 프로

〈표 3〉 예산량 변화

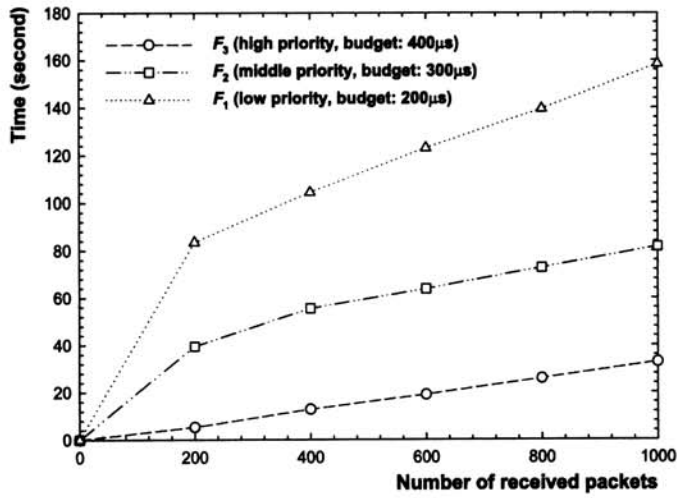
네트워크 흐름	(그림 5)에서의 예산 할당량	(그림 6)에서의 예산 할당량	증감
F_1	300μs	200μs	-100μs
F_2	300μs	300μs	0μs
F_3	300μs	400μs	+100μs



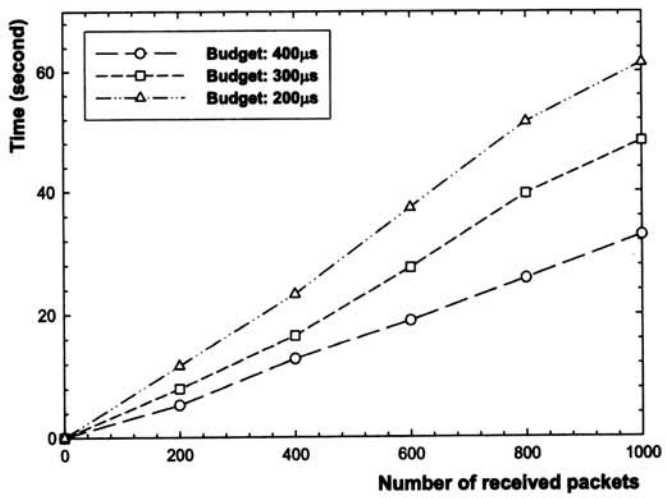
(그림 4) 리눅스 인터럽트기반 선착순방식에서의 측정결과



(그림 5) 제안하는 기법에서의 측정결과



(그림 6) 제안하는 기법에서 예산량 변경 후 측정결과



(그림 7) 하나의 네트워크 세션에서 예산량 변화에 따른 패킷 처리율 변화

토콜 처리 기법에서는 제어하기 어려웠던 QoS 격리 및 조절을 가능케 할 수 있다는 것을 알 수 있으며 이러한 결과는 제안하는 기법이 QoS 제공에 효과적으로 사용될 수 있음을 보여준다.

5. 결 론

본 논문에서는 bottom-half 메커니즘을 이용한 우선순위와 예산 기반의 새로운 네트워크 프로토콜 처리 기법을 제안하였으며 최대 패킷처리 시간에 대한 분석방법을 소개하였다. 제안하는 기법은 bottom-half 에서 동작하는 프로토콜 처리 핸들러를 각각의 네트워크 흐름에 할당하여 프로토콜 처리를 수행한다. 각각의 핸들러는 우선순위에 기반하여 스케줄링되며 자신에게 할당된 프로토콜 처리 실행예산 내에서 패킷을 처리한다. 이를 통해 우선순위 기반 프로토콜 처리방식의 부작용을 최소화할 수 있을 뿐만 아니라 각각의 네트워크 흐름에 대하여 효과적인 QoS 격리 및 조절이 가능하다. 아울러 수신된 모든 패킷들에 대하여 최대 패킷처리 시간을 한정(bound)시킬 수 있다. 향후 CAN(control area network) 또는 Bluetooth와 같은 프로토콜에서도 본 연구결과를 적용할 계획이며 다양한 CPU 스케줄링 정책 아래에서 패킷 처리율(throughput), 패킷처리 응답성 등 다양한 분석을 시도할 계획이다. 아울러 QoS 보장을 위한 동적 실행예산 및 우선순위 할당방법에 대해서도 향후 연구과제로 진행할 예정이다.

참 고 문 헌

[1] Chandramohan A. Thekkath, Thu D. Nguyen, Evelyn Moy, and Edward D. Lazowska, "Implementing network protocols at user level," Applications, Technologies, Architectures, and Protocols for Computer Communication, 1993.

[2] Fred Kuhns, Douglas C. Schmidt, and David L. Levine, "The Design and Performance of a Real-time I/O Subsystem," in Proceedings of the Fifth IEEE Real-Time Technology and Applications Symposium, 1999.

[3] Ashish Mehra, A. Indiresan, and K.G. Shin, "Structuring Communication Software for Quality-of-Service Guarantees," in Proceedings of the 17th IEEE Real-Time Systems Symposium, 1996.

[4] R. Gopalakrishnan and Gurudatta M. Parulkar, "Efficient User-Space Protocol Implementations with QoS Guarantees Using Real-Time Upcalls," IEEE/ACM Transactions on Networking, Vol.6, No.4, pp.374-388, Aug., 1998.

[5] R. Braden, D. Clark, and S. Shenker, "Integrated services in the Internet architecture: An overview," IETF, RFC 1633, Jun., 1994.

[6] S. Blake, D. Black, M. Calson, E. Davies, Z. Wang, and W. Weiss, "An architecture for differentiated services," RFC 2475, Dec., 1998.

[7] J.C. Mogul and K.K. Ramarkrishnam, "Eliminating Receive Livelock in an Interrupt-Driven Kernel," ACM Transactions on Computer Systmes, Vol.15(3), pp.217-252, 1997.

[8] D.D. Clark, V. Jacobson, J. Romkey, and H. Salwen, "An analysis of TCP processing overhead," IEEE Communications Magazine, Vol.27(6), pp.23-29, Jun., 1998.

[9] P. Druschel and G. Banga, "Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems," in Proceedings of the First Symposium on Operating Systems Design and Implementation, USENIX Association, pp.261-275, Oct., 1996.

[10] C.L Liu and J.W. layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," Journal of the ACM, Vol.20(1), pp.46-61, 1973.

[11] Chris Maeda and Brian N. Bershad, "Protocol Service Decomposition for High-Performance Networking," Symposium on Operating Systems Principles, pp.244-255, 1993.

[12] C. Lee, K. Yoshida, C. Mercer, and R. Rajkumar, "Predictable Communication Protocol Processing in Real-Time Mach," in Proceedings of Second Real-Time Technology and Applications Symp., June, 1996.

[13] A. Edwards, G. Watson, J. Lumley, D. Banks, C. Clamvokis, and C. Calton, "User-Space Protocols Deliver High Performance to Applications on a Low-Cost Gb/s LAN, Proc. ACM SIGCOMM, pp.14-24, London, Aug., 1994.

[14] V. Buch, T. von Eicken, A. Basu, and W. Vogels, "U-Net: A User Level Network Interface for Parallel and Distributed Computing," Proc. ACM Symp. Operating Systems Principles, pp.40-53, Dec., 1995.

[15] C.W. Mercer and H. Tokuda, "Preemptibility in Real-Time Operating Systems," Proc, Real-Time Systems Symp., Dec., 1992.

[16] Yuting Zhang and Richard West, "Process-Aware Interrupt Scheduling and Accounting," in Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS), Dec., 2006.

[17] R. Gopalakrishnan and Gurudatta M. Parulkar, "Quality of Service Support for Protocol Processing Within Endsystems," in Proceedings of the 2nd International Workshop on Architecture and Protocols for High Performance, pp.179-198, 1995.

[18] Linux, available at http://www.kernel.org/

[19] M. Singh, P. Pradhan, and P. Francis, "MPAT: aggregate TCP congestion management as a building block for Internet QoS," in Proceedings of IEEE International Conference on Network Protocols, pp.129-138, Oct., 2004.

[20] H. Y. Hsieh and K.H. Kim and R. Sivakumar, "On Achieving Weighted Service Differentiation: an End-to-end Perspective," in Proceedings of IEEE IWQoS, Jun., 2003.

[21] Xia Gao, Gang Wu, and Toshio Miki, "End-to-end QoS

provisioning in mobile heterogeneous networks," IEEE Wireless Communications, Vol.11, No.3, Jun., 2004.
[22] R. Braden et al., "Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification," Internet RFC 2205, Sept., 1997.
[23] A.Venkataramani and R. Kokku and Mike Dahlin, "TCP NICE: A mechanism for background transfers," in Proceedings of the 5th Symposium on Operating Systems Design and Implementation, pp.329-344, Dec., 2002.

김 지 민

e-mail : jmkim@rtcc.hanyang.ac.kr
2003년 한양대학교 전자전기공학부(학사)
2005년 한양대학교 정보통신대학(공학석사)
2009년~현 재 한양대학교 전자컴퓨터통신공학과 박사과정
관심분야 : 실시간 시스템, 임베디드 소프트웨어, 멀티 프로세서용 운영체제, QoS(Quality of Service), RFID 미들웨어, 소프트웨어 공학 등

유 민 수

e-mail : msryu@hanyang.ac.kr
1995년 서울대학교 제어계측공학과(학사)
1997년 서울대학교 제어계측공학과(공학석사)
2002년 서울대학교 전기컴퓨터공학부(공학박사)
2003년~현 재 한양대학교 컴퓨터공학부 조교수
관심분야 : 실시간 시스템, 임베디드 소프트웨어, RTOS, 소프트웨어 공학 등