Journal of Korea Multimedia Society Vol. 8, No. 11, November 2005(pp. 1472-1482)

TOE를 위한 소켓 인터페이스의 구현

손 성 훈†

요 약

TOE (TCP/IP Offload Engine)는 부하가 많은 대규모 네트워크 서버에서 TCP/IP 프로토콜 처리의 부담을 줄이기 위해 고안된 하드웨어 장치이다. 본 논문에서는 TOE (TCP Offload Engine)를 사용하는 대규모 멀티미디어 서버를 위한 소켓 인터페이스 계층의 설계 및 구현에 대해 다룬다. 제안된 소켓 인터페이스 계층은 리눅스 운영체제 상에서 커널 모듈로 설계, 구현되었으며, BSD 소켓 계층과 INET 소켓 계층 사이에 존재하면서 응용 프로그램의 소켓 관련 요청을 TOE나 기존 INET 소켓 계층으로 전달하는 역할을 한다. 본 논문에서 제안한 소켓 인터페이스는 소켓을 통해 TOE를 사용하는 응용 프로그램에 대해서 모든 표준 소켓 입출력 API와 파일 입출력 관련 API를 그대로 제공하고, 기존 응용 프로그램들에 대해서도 수정 없이 TOE의 기능을 그대로 사용할 수 있는 바이너리 수준의 호환성을 제공하며, 한 시스템에서 TOE와 이더넷 NIC을 동시에 사용할 수 있게 된다.

An Implementation of Socket Interface for TOEs

Sunghoon Son†

ABSTRACT

In this paper, we propose a socket interface layer for large-scale multimedia servers that adopt TCP/IP Offload Engines (TOE). In order to provide legacy network applications with binary level compatibility, the socket interface layer intercepts all socket-related system calls to forward to either TOE or legacy TCP/IP protocol stack. The layer is designed and implemented as a kernel module in Linux. The layer is located between BSD socket layer and INET socket layer, and passes the application's socket requests to INET socket layer or TOE. The layer provides multimedia servers and web servers with the following features: (1) All standard socket APIs and file I/O APIs that are supported (2) Support for binary level compatibility of existing socket programs (3) Support for TOE and legacy Ethernet NICs at the same time.

Key words: Multimedia Servers(멀티미디어 서버), TCP/IP, TCP Offload Engine(TCP 오프로드 엔진), Socket Interface(소켓 인터페이스)

1. 서 론

전통적인 네트워크 서버에서 TCP/IP 프로토콜은 흔히 운영체제의 통신 서브 시스템 모듈의 형태로 구현되어 왔다. 이는 기존의 네트워크 서버의 구조에서는 TCP/IP 프로토콜 처리의 대부분이 소프트웨어적으로 처리되어 왔음을 의미한다. 네트워크 대역폭이 기가비트급 이상으로 늘어나고 한 네트워크 서버가 동시에 처리해야 하는 TCP 연결의 개수가 급격히 늘어남에 따라, 서버의 메인 프로세서에 의존한 통신 프로토콜의 소프트웨어적인 처리 방식은 성능 상의 한계를 드러내고 있다. 실제로 Apache 웹 서버를 수행하는 서버의 경우, 전체의 71%의 시간을 TCP 프로토콜의 처리에 사용되는 것으로 조사되었다[1].

※ 교신저자(Corresponding Author) : 손성훈, 주소 : 서울시 종로구 홍지동 7(110-743), 전화 : 02)2287-5137, FAX : 02)396-5704, E-mail : shson@smu.ac.kr
접수일 : 2005년 3월 4일, 완료일 : 2005년 6월 17일
† 상명대학교 소프트웨어학부 전임강사

즉, 네트워크 서버의 동작에서 프로세서 수행의 많은 부분을 TCP/IP 프로토콜 처리에 할애하게 됨으로써, 정작 네트워크 서버가 수행해야 하는 응용 프로그램은 많은 처리를 받지 못하게 되어, 네트워크 대역폭이 늘어난 만큼의 전체적인 성능 향상을 기대할 수 없게 되는 것이다.

TOE(TCP/IP Offload Engine)는 이처럼 대규모 네트워크 서버에서 TCP/IP 프로토콜 처리시 발생하는 성능 상의 문제점들을 해결하기 위해 고안된 하드웨어 장치이다. 기존의 일반 NIC(Network Interface Card)의 경우, 단지 이더넷 프로토콜만을 하드웨어적으로 구현하고, IP 프로토콜이나 TCP/UDP 프로토콜은 소프트웨어적으로 구현하여 사용하게 된다. 반면 TOE의 경우, 이더넷 프로토콜 뿐만 아니라, 상위의 IP 프로토콜이나 TCP/UDP 프로토콜까지도 TOE 내에 하드웨어적으로 구현함으로써, 대부분의 TCP/ IP 프로토콜의 처리를 하드웨어가 담당하도록 하고 있다. 따라서 TOE를 사용하는 경우 네트워크 서버의 호스트 프로세서는 통신 프로토콜 모듈의 수행에 따른 부담에서 벗어나 좀 더 많은 프로세서 수행 사이클을 응용 프로그램에 할애할 수 있게 되어, 네트워크 대역폭이 증가한 만큼의 효과를 제대로 누릴 수 있게 된다.

최근 이러한 TOE에 대한 많은 연구들이 진행되고 있으며[2-4], TOE 카드를 채용한 대규모의 네트워크 서버들이 개발되고 있다[1,5]. TOE를 채용한 네트워크 서버의 경우 앞서 언급한 성능 상의 문제는 개선하였으나, 네트워크 응용 프로그램의 수행에 대한 호환성 측면에서는 또다른 문제를 가지고 있다. 대부분의 네트워크 응용 프로그램들은 표준 BSD 소켓 API를 사용하여 TCP/IP를 이용한 통신 기능을 구현하고 있다. 그러나, 대부분의 TOE 카드는 기본적인 장치 구동기(device driver)와 함께, TOE의 오프로드 TCP/IP를 사용할 수 있도록 표준 BSD 소켓 API가 아닌 별도의 통신 API를 제공한다. 따라서, 네트워크 응용 프로그램이 TOE의 기능을 사용하기 위해서는 표준 소켓 API를 사용하는 대신, 개별 TOE 카드에 의존적인 API를 사용해야 한다. 이는 기존의 BSD 소켓 인터페이스를 사용하는 네트워크 응용 프로그램이 TOE 카드를 장착한 시스템에서 동작하기 위해서는 응용 프로그램의 소스 코드를 수정해야 함을 의미한다.

표준 BSD 소켓 인터페이스를 그대로 사용하도록 설계된 일부 TOE 카드의 경우에도 기존 네트워크 응용 프로그램에 대해 완전한 바이너리 수준의 호환성은 제공하지 못한다. 예를 들어 그림 1의 경우와 같이 일반적인 NIC과 표준 소켓 인터페이스를 제공하는 TOE가 한 서버에 동시에 장착되어 있는 경우를 가정해보자. 즉, 기능 상으로는 동일한 두 개의 TCP/IP 프로토콜 스택이 다른 형태로 한 시스템 내에 구현되어 있는 경우이다. 대부분의 전형적인 BSD 소켓 계층의 구현에 따라, TOE를 위한 소켓 계층은 표준 BSD 소켓 계층 아래에 하나의 프로토콜 패밀리로서 존재하게 된다[6-8]. 물론 일반적인 NIC(과 기존 TCP/IP 프로토콜 스택)을 위한 소켓 계층도 BSD 소켓 계층 아래에 나란히 존재하고 있으며, 두 소켓 계층은 서로 다른 프로토콜 패밀리 식별자에 의해 구분되어 사용된다(네트워크 응용 프로그램은 socket() 시스템 호출의 인자로 특정 프로토콜 패일리 식별자를 명시함으로써 구분하게 된다). 이러한 경우 새로이 작성되는 네트워크 응용 프로그램이 표준 BSD 소켓 API를 통해 TOE를 사용할 수 있는 것은 사실이나, 기존의 네트워크 응용 프로그램이 TOE 상에서 동작하려면 소켓 생성 시에 명시하는 네트워크 패밀리 식별자를 TOE에 맞도록 수정해야만 한다.

본 논문에서는 TOE를 위한 소켓 모듈인 SOP (Socket over PCI[1]) 모듈에 대해 소개한다. 기본적으로 SOP는 TOE에 대한 표준 BSD 소켓 인터페이스를 제공하는 모듈이다. 특히 SOP 모듈은 소켓 응용

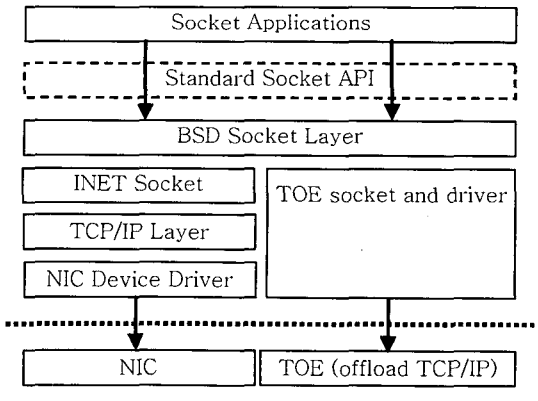

그림 1. 일반 NIC- TCP/IP와 TOE-오프로드 TCP/IP

1) 대부분의 TOE들이 PCI 인터페이스 카드 형태고, 이 경우 SOP 모듈이 PCI 드라이버 모듈 바로 위에서 동작하게 된다는 점에서 기인한 이름이다.

프로그램이 일반 네트워크 인터페이스 카드를 사용하는 것과 동일한 방식으로 TOE를 사용할 수 있도록 해준다. 즉, 소켓 응용 프로그램은 기존 TCP/IP 프로토콜 스택을 사용하기 위해 지정하는 AF_INET 프로토콜 패밀리 식별자를 그대로 사용해 소켓을 생성하고 TOE의 오프로드 TCP/IP 프로토콜 스택을 사용할 수 있게 된다. 이를 위해 SOP 계층은 응용 프로그램이 소켓 관련 시스템 호출을 요청할 때 마다, 해당 호출이 기존 TCP/IP 프로토콜 스택을 위한 것인지, TOE의 오프로드 TCP/IP 스택을 위한 것인지를 판단하여 적절한 소켓 계층으로 이 요청을 전달함으로써, 기존의 소켓 프로그램들이 일반 NIC과 TOE에 대한 구분없이 투명하게 TCP/IP 프로토콜을 사용하는 것이 가능하다.

SOP 모듈은 리눅스 운영체제 상에서 커널 모듈(kernel module)로 설계 및 구현되었다[9]. SOP 모듈은 구체적으로 다음과 같은 기능들을 제공한다.

- 응용 프로그램이 TOE 카드를 사용함에 있어서 기존 리눅스 운영체제가 제공하는 모든 표준 소켓 API와 파일 입출력 API들이 그대로 지원한다.
- 기존 소켓 프로그램들에 대해 바이너리 수준의 호환성이 제공한다.
- 기존의 이더넷 네트워크 인터페이스 카드와 여러 장의 TOE 카드들을 한 시스템에서 동시에 사용할 수 있다.
- 다양한 형태의 TCP/IP 오프로드 엔진들로 쉽게 확장할 수 있다.
- TOE 카드의 설정/관리를 위해 기존에 네트워크 설정에 쓰이는 유틸리티들을 그대로 사용할 수 있다.
- SOP 모듈은 기존 커널 코드의 수정 없이 커널 모듈로만 구현한다.

본 논문의 구성은 다음과 같다. 2절에서는 SOP 계층의 전체 구조를 소개하고, 각 구성 요소들 간의 동작을 기술한다. 3절에서는 SOP 계층의 상세 설계에 대해 다룬다. 우선 SOP 계층의 주요 자료 구조 등을 설명하고, 대표적인 소켓 관련 시스템 호출 별로 SOP 계층이 이들을 처리하는 알고리즘에 대해 다루고, 마지막으로 4절에서는 결론을 맺는다.

2. SOP 계층

그림 2는 리눅스 운영체제 커널 내에서 BSD 소켓

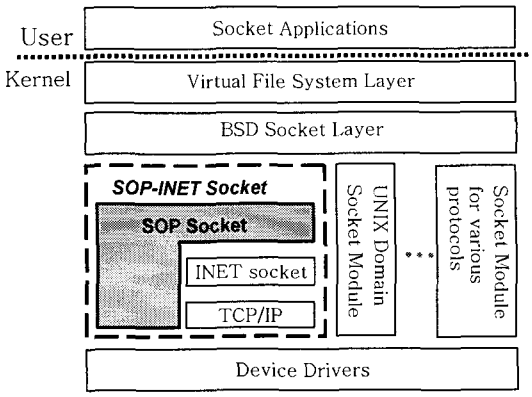

그림 2. SOP 계층

계층, 각종 프로토콜 스택 모듈, 그리고 SOP 모듈 간의 관계를 나타내고 있다. 보통 응용 프로그램이 소켓 인터페이스를 통해 새로운 통신 프로토콜을 사용할 수 있도록 하려면, 해당 통신 프로토콜 모듈을 BSD 소켓 계층 아래에 하나의 프로토콜 패밀리로서 등록해야 한다. 이 과정에서 BSD 소켓 계층은 하위의 각 프로토콜 스택 모듈마다 유일한 프로토콜 패밀리 식별자를 부여함으로써 이들을 구분한다. 소켓 응용 프로그램이 특정 통신 프로토콜의 프로토콜 패밀리 식별자와 함께 BSD 소켓 계층을 호출하면, BSD 소켓 계층은 제공된 프로토콜 패밀리 식별자를 통해 응용 프로그램이 요청한 호출을 어느 프로토콜 모듈로 전달해야 하는지를 결정하게 된다.

본 논문에서는 기존 TCP/IP 스택과 TOE의 오프로드 TCP/IP 스택 간에 공통의 프로토콜 패밀리 식별자 (AF_INET)를 공유하도록 설계되었기 때문에, 기존의 BSD 소켓 계층만으로는 응용 프로그램의 요청을 어느 TCP/IP 스택으로 전달해야 하는지 판단할 수 없다. 따라서 BSD 소켓 계층과 두 TCP/IP 스택 중간에 SOP 소켓 계층을 두어, 소켓 프로그램의 요청이 어느 프로토콜 스택에 해당하는 것인지를 판단하도록 한다. 이를 위해 SOP 소켓 계층은 새로운 소켓을 생성할 때, 내부적으로 두 개의 소켓 자료 구조, 즉 기존 TCP/IP 스택을 위한 소켓과 오프로드 TCP/IP 스택을 위한 소켓을 만들게 된다. SOP 계층은 두 소켓 자료 구조 중 최종적으로 어느 쪽이 실제 통신에 사용되는지 판단할 수 있을 때까지 두 개의 소켓 자료 구조를 유지한다. 이후 소켓 프로그램이 accept()나 connect()와 같은 시스템 호출을 수행하게 되면, SOP 계층은 이 호출을 바탕으로 두 개의

소켓 자료 구조 중 어느 쪽이 실제로 사용될 지 결정하게 된다.

실제 구현된 프로토타입에서 SOP 모듈은 호스트 측 모듈, (TOE) 타겟 측 모듈, 그리고 호스트와 타겟 간의 통신 모듈로 구성된다. 호스트 모듈은 앞서 언급한 기능을 제공하는 호스트 운영체제의 커널 모듈이다. 타겟 모듈은 TOE 카드 내부의 소프트웨어 모듈로, 호스트 측 모듈이 보내온 요청을 TOE 카드 내의 오프로드 TCP/IP 스택으로 전달하는 역할을 한다. 통신 모듈은 호스트 모듈과 타겟 모듈 간에 명령, 파라미터, 리턴값 등을 주고 받는데 사용되는 모듈이다. 그림 3은 관련 자료 구조와 함께 세 가지 모듈 간의 동작을 나타내고 있다. 그림의 좌측은 호스트 측의 SOP 관련 커널 내부의 구조이고, 우측은 TOE 카드의 내부 구조를 나타낸다. 예를 들어, 소켓 응용 프로그램이 connect() 시스템 호출을 요청한 경우, BSD 소켓 계층을 거쳐 SOP 계층의 sop_stream_connect() 함수가 일단 호출된다. Sop_stream_connect() 함수는 응용 프로그램이 요청한 시스템 호출이 기존 TCP/IP 스택을 위한 것인지, 오프로드 TCP/IP 스택을 위한 것인지를 판단하게 된다. 일반 TCP/IP 스택에 해당하는 호출이면 기존의 connect 연산을 처리하는 함수인 inet_stream_connect()를 호출하게 되며, 오프로드 TCP/IP 스택을 위한 것이면 해당하는 connect 의 인자들을 통신 모듈을 사용해서 타겟 모듈로 전달하여, TOE 내에서 connect 연산이 처리될 수 있도록 한다. 통신 모듈과 타겟 모듈은 다양한 TOE 카드 구현 방법에 따라 달라질 수 있으며, 본 논문에서는 주로 호스트 측 SOP 모듈에 대해서 중점적으로 다룬다.

3. SOP의 상세 설계

본 절에서는 SOP 계층에 대한 상세 설계를 다룬다. 우선 SOP 계층의 주요 지료 구조를 소개하고, SOP 계층이 소켓 관련 시스템 호출들을 어떤 방식으로 처리하는지 다룬다.

3.1 자료 구조

SOP 계층은 리눅스 운영체제 상에서 커널 모듈로 구현되었기 때문에, 기존 커널에 구현되어 있는 소켓 계층이나 TCP/IP 프로토콜 스택에 대한 수정이 필요 없으며, 필요시 시스템 동작 중에 동적으로 모듈을 추가, 삭제할 수도 있다. 그림 4는 SOCK_STREAM 타입의 소켓의 예를 통해 하나의 소켓과 관련한 SOP 소켓 계층의 주요 자료 구조들을 나타내고 있다. 그림에서 보는 바와 같이 SOP 계층은 SOP 소켓 생성 함수(struct net_proto_family sop_ family_

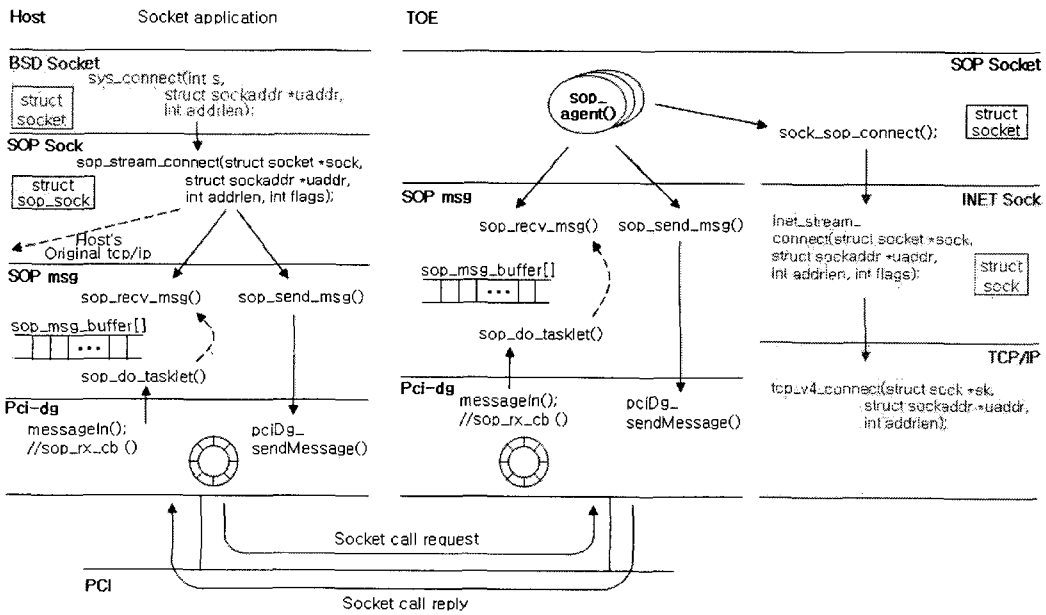

그림 3. SOP 모듈의 동작

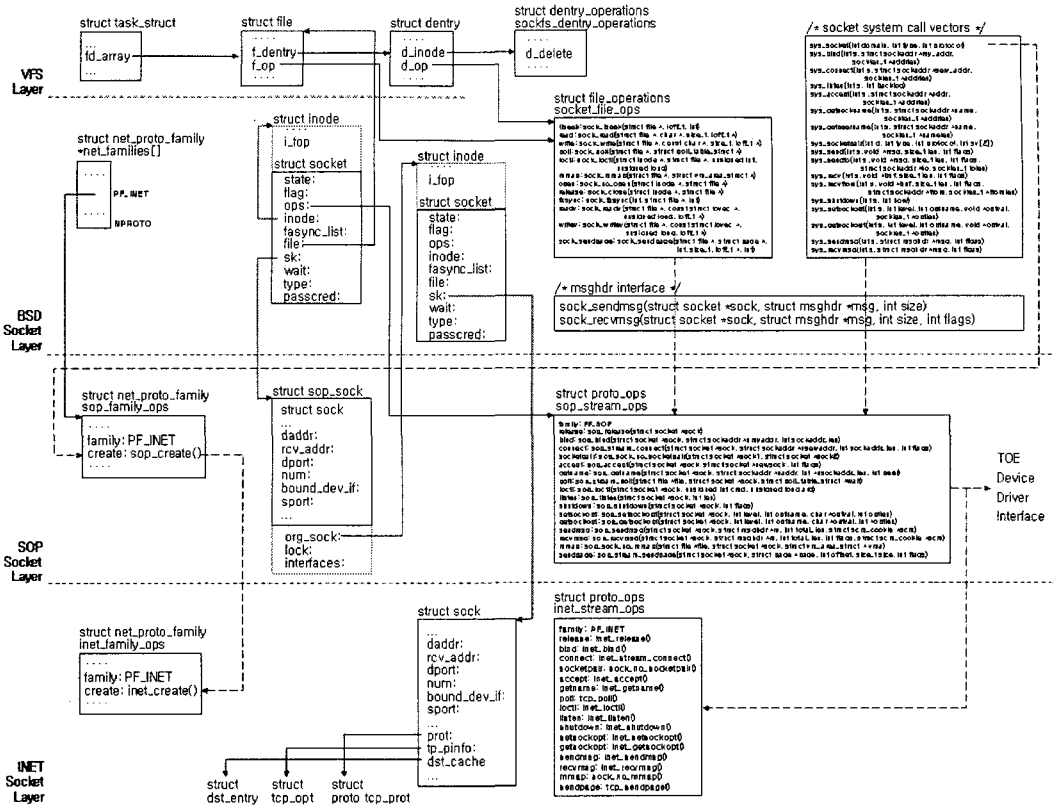

그림 4. SOP 계층의 자료구조

ops 내의 sop_create()), SOP 소켓(struct sop_sock), 그리고 일련의 SOP 계층의 STREAM 소켓 처리 함수들(struct proto_ops sop_stream_ops)로 이루어진다.

SOP 소켓 생성 함수는 새로운 SOP 소켓을 생성하고, 이 소켓을 다른 자료 구조들과 연결하며, TOE 카드에 새로운 소켓 생성과 관련한 초기화 명령을 보내는 등의 역할을 한다.

SOP 소켓은 기존의 INET 소켓(struct sock)의 확장된 형태를 가진다. 기본적으로 SOP 소켓은 기존 INET 소켓이 가지고 있는 모든 정보를 포함하고 있으며, 추가로 이 소켓이 어느 TOE 카드와 연관되어 있는지에 대한 정보(sop_sock의 ps 필드) 등을 가지고 있다.

SOP 계층의 STREAM 소켓 처리 함수 집합은 SOCK_STREAM 타입의 SOP 소켓에 대한 여러 가지 연산들로 이루어진다. 이 연산들은 응용 프로그램이 소켓 관련 시스템 호출을 수행함에 따라, BSD 소켓 계층의 해당 소켓 연산(struct file_operations socket_file_ops)에 의해 불리워진다. 일단 하나의 BSD 소켓 연산에서 이 소켓 연산 함수를 호출하게 되면, 해당 함수는 기존의 INET 소켓 계층의 STREAM 함수를 호출하거나, TOE 카드에 적절한 명령을 전달하게 된다. SOCK_DGRAM 타입의 소켓에 대해서도 비슷한 동작이 이루어진다.

최초로 SOP 소켓 모듈이 초기화될 때, SOP 모듈은 기존 BSD 소켓 계층의 프로토콜 모듈 별 소켓 생성 함수의 배열(struct net_proto_family *net_families[])에 등록되어 있던 INET 소켓 생성 함수를 SOP 계층의 소켓 생성 함수(struct net_proto_family sop_family_ops)로 대체한다. 이후 응용 프로그램이 새로운 소켓의 생성을 요청하면, BSD 소켓 계층은 일단 기본적인 BSD 소켓(struct socket)을 생성한 후, 새로 등록된 SOP 소켓 생성 함수를 호출하게 된다. SOP 소켓 생성 함수는 우선 기존 커널 코드의 INET 소켓 생성 함수를 사용하여 하나의 INET 소켓을 생성함과 동시에 SOP 소켓을 함께 만들게 된다. 또한 SOP 소켓 생성 함수는 각 TOE 카드

에 새로 생성된 소켓을 위한 초기화 작업을 수행하도록 명령을 보낸다.

이처럼 SOP 계층이 하나의 socket() 시스템 호출에 대해 내부적으로 두 개의 소켓 자료 구조를 만드는 이유는, 소켓 생성 시점에서 응용 프로그램이 기존 TCP/IP 프로토콜 스택을 사용할지, 오프로드 TCP/IP 프로토콜 스택을 사용할 지 알 수 없기 때문이다. SOP 계층은 두 소켓 구조 중 하나가 더 이상 필요없다고 판단될 때까지는 두 소켓 자료 구조를 모두 유지한다. 이러한 판단은 소켓 생성 후, 응용 프로그램이 요청하는 일련의 소켓 관련 시스템 호출을 처리하는 과정에서 판단할 수 있다. 예를 들어 SOP 계층의 bind 함수인 sop_bind()는 사용자의 bind() 시스템 호출로부터 넘겨받은 주소 인자로부터 이후에 어느 프로토콜 스택이 사용될 지를 판단할 수 있다. 만일 바인드 하고자 하는 주소가 기존 NIC에 해당하는 것이라면, SOP 계층은 기존 TCP/IP 프로토콜 스택의 bind 함수만을 호출하게 되며, 이러한 판단 결과를 SOP 소켓에 기록하게 된다. 이후 이 소켓에 대한 시스템 호출에서는 이를 참조하여 기존 TCP/IP 프로토콜 스택에 대해서만 연산이 이루어지게 된다. 반면 bind 주소가 TOE에 해당하는 경우, 이후 이 소켓과 관련된 모든 프로토콜 처리는 해당 TOE의 오프로드 TCP/IP 스택으로만 전달된다.

3.2 소켓 시스템 호출의 처리

본 절에서는 SOP 계층에 대한 기본 개념과 자료 구조를 기반으로 주요 소켓 관련 시스템 호출들이 어떤 식으로 동작하는지에 대해 설명한다. SOP 계층의 각 소켓 시스템 호출들은 기존의 소켓 관련 시스템 호출이 가지고 있는 의미들을 최대한 유지하도록 설계되었다.

3.2.1 SOCKET

Socket 은 소켓 응용 프로그램이 최초로 소켓을 생성하는 시스템 호출이다. 새로운 SOP 소켓의 생성에 대한 처리는 3.1절에서 다룬 바와 같다.

3.2.2 BIND

하나의 TCP/IP 연결은 두 개의 IP 주소/포트 번호 쌍으로 표현되며, bind() 시스템 호출은 소켓에 로컬 주소와 포트 번호를 할당하는 역할을 한다. SOP 계층의 bind 함수인 sop_bind()는 사용자로부터 전달된 로컬 주소를 체크하여, 이 주소가 로컬 NIC에 연관된 주소이면 기존 프로토콜 스택의 bind 함수인 inet_bind()를 호출하고, TOE 중 하나에 연관된 주소이면 해당 TOE 측의 bind 함수를 호출한다. 만일 사용자가 INADDR_ANY로 바인드 하고자 한다면, inet_bind()와 각 오프로드 TCP/IP 스택의 bind 함수를 모두 호출한다. 포트 번호의 경우, 만일 사용자가 포트 번호를 지정한 경우에는, 해당 번호가 사용 가능한 경우에 이를 할당한다. 사용자가 포트 번호 0을 지정한 경우, 사용되지 않는 포트 번호를 자동적으로 할당한다.

한 소켓에 대한 bind 연산이 성공적으로 완료되면, SOP 계층은 이후의 소켓 시스템 호출을 어느 프로토콜 스택이 수행해야 하는지를 쉽게 판단할 수 있다. SOP 계층은 이후의 소켓 시스템 호출을 위해 bind의 마지막 단계에서 해당 소켓이 어느 로컬 주소에 바인드 되어 있는지를 sop_sock 구조체에 저장해 놓는다.

3.2.3 LISTEN

Listen() 시스템 콜은 TCP 서버 측에서 호출하는 함수로, 아직 연결이 이루어지지 않은 소켓에 대해서 앞으로 클라이언트 측의 연결 요청을 받아들이겠다는 것을 선언하는 역할을 한다. SOP 계층의 listen 연산을 처리하는 함수인 sop_listen() 함수는 listen() 시스템 콜이 호출되는 시점에서의 소켓의 바인드 상태를 판단하여 기존 TCP/IP 스택과 오프로드 TCP/IP 스택 중 어느 TCP/IP 스택에 대해서 listen 연산을 요청할 지를 결정한다. 즉, 앞서 bind 에서 sop_sock 구조체에 기록된 로컬 주소를 바탕으로, 이 소켓과 바인드 된 로컬 주소에 해당하는 TCP/IP 스택에 대해 listen 연산을 호출하게 된다.

아직 바인드 되어 있지 않은 채 listen() 시스템 호출이 요청된 경우에는 먼저 소켓의 로컬 주소를 INADDR_ANY로 바인드 시킨다. 앞서 언급한 바와 같이 소켓이 INADDR_ANY로 바인드 되면 모든 TCP/IP 스택으로 들어오는 접속 요청을 다 받아들이겠다는 것을 의미한다. 따라서 SOP 계층은 INADDR_ANY로 바인드 시킨 후 모든 TCP/IP 스택에 대해서 listen 연산을 요청한다.

3.2.4 ACCEPT

Accept() 시스템 호출은 흔히 TCP 서버에 의해

불리워지며, 대기 중인 연결 요청을 큐에서 꺼내 클라이언트 측과 연결을 생성하게 된다. Accept 연산은 이미 바인드 된 소켓에 대해서만 수행할 수 있다. 소켓이 이미 특정 로컬 주소에 바인드 되어 있으면, SOP 계층은 그 로컬 주소와 연관된 TCP/IP 스택에 대해서 accept 연산을 요청하게 된다. Listen 의 경우와 마찬가지로, 소켓이 INADDR_ANY 주소에 바인드 되어 있는 경우에는 각 TCP/IP 스택의 accept 연산을 모두 호출한다. 소켓이 특정 로컬 주소에 바인드된 경우에는 해당 로컬 주소와 관련된 TCP/IP 스택의 accept 연산을 호출하게 된다. 하위 TCP/IP 스택에 요청한 accept 연산이 성공적으로 처리되면, SOP 계층은 새로운 연결에 대한 추가의 SOP 소켓을 만든다.

일반적으로 블록킹 입출력 모드의 소켓의 경우, accept 연산의 처리는 이를 요청한 프로세스를 대기 상태로 만들 수 있다. 따라서, SOP 계층에서 두 개 이상의 TCP/IP 스택에 순차적으로 accept 를 요청하는 것은 문제가 있다. 따라서 이 경우 SOP 계층은 별도의 커널 쓰레드 (kernel thread)를 생성하고, 이 커널 쓰레드들로 하여금 대신 accept를 요청하게 함으로써, 동시에 여러 개의 accept 요청이 가능하도록 하였다. 블록킹 모드의 소켓이 아니거나 accept를 요청해야하는 TCP/IP 스택이 유일한 경우에는 이러한 부가적인 커널 쓰레드의 생성은 필요없게 된다.

3.2.5 CONNECT

TCP 소켓의 경우, connect() 시스템 호출은 연결된 소켓을 생성한다. UDP 소켓의 경우에는 단지 이후의 send 연산에서 사용될 대상 주소를 지정하는 역할을 한다. Connect 연산에 대한 SOP 계층의 처리 역시 소켓의 바인드 상태에 따라 다르다. TCP 소켓의 경우, 아직 소켓이 어느 로컬 주소에도 바인드되어 있지 않다면, 우선 소켓의 로컬 주소를 INADDR_ANY 로 바인드한 후, 목적지 주소에 대한 경로가 존재하는 각 TCP/IP 스택에 대해 connect 연산이 성공할 때까지 차례로 connect 연산을 요청한다. 이미 특정 로컬 주소에 바인드된 TCP 소켓의 경우에는, 단순히 해당 프로토콜 스택의 connect 연산을 호출하게 된다. UDP 소켓의 경우, SOP 계층은 단순히 대상 주소에 대한 경로가 존재하는 TCP/IP 스택 중 하나에 대해 connect 연산을 호출하게 된다.

3.2.6 SENDMSG

소켓 응용 프로그램이 send(), sendto(), sendmsg() 등과 같은 시스템 호출을 요청하면 SOP 계층의 sendmsg 연산이 수행된다. SOP 계층의 sendmsg 연산에서는 우선 해당 소켓의 바인드 상태를 확인하고, 이미 바인드 된 로컬 주소가 있는 경우 해당 로컬 주소와 연관된 TCP/IP 스택으로 전송할 데이터를 전달한다(즉, 해당 TCP/IP 스택의 sendmsg 연산을 호출한다). 만일 소켓이 INADDR_ANY로 바인드 되어 있다면, SOP 계층은 각 TCP/IP 스택에 대해 성공할 때까지 차례로 sendmsg 연산을 호출한다. 만일 바인드 되어 있지 않은 소켓이면, 먼저 임의의 새로운 포트를 할당하고, 이 포트를 사용하여 모든 하위 TCP/IP 스택에 bind 연산을 수행한 후, 각 TCP/IP 스택에 대해 sendmsg 연산이 성공할 때까지 차례로 호출한다.

SOP 소켓 구조체는 아직 바인드되지 않았거나,. INADDR_ANY로 바인드 된 소켓에 대해 가장 최근에 성공적으로 sendmsg를 수행한 목적지 주소와 포트 번호, TCP/IP 스택에 대한 정보를 저장하는 필드를 가지고 있다. 이 정보는 이후 동일한 목적지에 대해 sendmsg 가 반복 수행될 때 좀 더 신속한 수행을 위하여 사용된다.

3.2.7 RECVMSG

SOP 계층의 recvmsg 연산은 소켓 응용 프로그램이 recv, recvfrom, recvmsg 등의 시스템 호출을 요청했을 때 동작한다. SOP 계층은 특정 로컬 주소로 바인드 된 소켓에 대해 해당 주소와 연관된 프로토콜 스택의 recvmsg 연산을 호출한다. 소켓이 INADDR_ANY 주소로 바인드 된 경우, SOP 계층의 동작은 소켓의 입출력 모드가 블록킹 모드인지 아닌지에 따라 달라진다. Non blocking 모드의 recvmsg 연산의 경우, SOP 계층은 단순히 각 TCP/IP 스택의 recvmsg 연산을 차례로 호출하게 된다. 반면 블록킹 모드에서는 SOP 계층은 일정 시간 간격으로 모든 TCP/IP 스택의 recvmsg 연산을 성공할 때까지 차례로 non-blocking 모드로 호출한다.

3.2.8 GETNAME

Getname 연산은 주어진 소켓관 연관된 소스 또는 목적지 주소와 포트 번호를 반환한다. SOP 계층은

역시 소켓의 현재 바인드 상태를 보고 어느 TCP/IP 스택의 getname 연산을 호출할 것인지 판단한다. 소켓이 TOE와 연관된 로컬 주소에 바인드 되어 있는 경우, SOP 계층은 해당 TOE의 TCP/IP 스택에 getname 연산을 요청한다. 그렇지 않은 경우, INET 소켓 계층에 getname 연산을 요청한다.

3.2.9 IOCTL

Ioctl()은 네트워크 장치의 설정과 관련된 다양한 일을 하는 함수이다. SOP 계층의 ioctl 연산은 응용 프로그램이 제시한 명령과 관련 인자들을 바탕으로 요청한 ioctl 연산이 어느 TCP/IP 스택에서 처리되어야 하는 것인지 결정하는 역할을 한다. 이러한 방식은 SOP 계층의 다른 연산들의 동작과 대체로 유사하나, 처리해야 하는 하위 명령어에 따라 각기 다른 방식으로 동작하기도 한다.

3.2.10 POLL

Poll() 소켓과 관련된 이벤트의 발생을 기다리는 시스템 호출이다. 만일 SOP 소켓이 TOE와 연관된 로컬 주소에 바인드 되어 있는 경우, SOP 계층은 해당 TOE의 TCP/IP 스택에 대해 poll 연산을 요청한다. 그 외의 경우에는 모든 TCP/IP 스택의 poll 연산을 요청한다.

3.2.11 SETSOCKOPT

Setsockopt는 소켓에 대한 각종 소켓 옵션을 수정하는 연산이다. SOP 소켓이 특정 로컬 주소에 바인드 되어 있는 경우, SOP 계층은 단순히 해당 TCP/IP 스택의 setsockopt 연산을 요청한다. 그 외의 경우에는 각 TCP/IP 스택에 대해 setsockopt 연산을 차례로 호출한다.

3.2.12 GETSOCKOPT

Getsockopt는 각종 소켓 옵션을 읽는 연산이다. 소켓이 특정 로컬 주소에 바인드 되어 있는 경우, SOP 계층은 단순히 해당 TCP/IP 스택의 getsockopt 연산을 요청한 후, 그 결과를 응용 프로그램에게 반환한다. 그 외의 경우에는 기존 TCP/IP 스택의 연산을 호출한 후 그 결과를 하게 된다 (즉, 특정 소켓에 대한 소켓 옵션은 INET 프로토콜 스택의 것이 대표한다고 가정함)

3.2.13 SHUTDOWN

SOP 계층의 shutdown 연산은 소켓의 양방향 (full-duplex) 연결 중 하나를 끊는 기능을 한다. 소켓이 특정 로컬 주소에 바인드 되어 있는 경우, SOP 계층은 단순히 해당 TCP/IP 스택의 shutdown 연산을 요청한다. 그 외의 경우에는 각 TCP/IP 스택에 대해 shutdown 연산을 차례로 호출한다.

3.2.14 RELEASE

SOP 계층의 release 연산은 소켓을 닫는 기능을 제공한다. 이는 단순히 모든 TCP/IP 스택의 release 연산을 호출하는 것으로 이루어진다.

4. 성능 평가

본 논문에서 제안한 SOP 모듈의 사용으로 커널 소켓 계층에는 '프로토콜 스택의 동적인 선택'이라는 기능이 추가되었다. 이러한 새로운 기능의 추가는 작게는 소켓 관련 시스템 호출 수준에서, 크게는 네트워크 서버 전체의 성능에 영향을 미칠 수 있다. 본 장에서는 SOP 모듈의 동작이 시스템 성능에 미치는 영향을 정량적으로 분석하고, 이를 통해 SOP 모듈이 네트워크 서버의 성능에 미치는 영향이 작도록 효율적으로 설계, 구현되었음을 보인다.

우선 SOP 모듈이 개별 소켓 시스템 호출의 성능에 어떤 영향을 미치는지를 측정하였다. 여러 소켓 관련 시스템 호출 중 send 시스템 호출은 웹 서버와 같은 네트워크 서버 응용의 성능에 중요한 역할을 한다. 표 1은 SOP 모듈의 사용 여부에 따른 send 시스템 호출의 지연 시간(latency)을 측정한 결과이다. 지연 시간은 각 경우마다 10,000번의 send 호출의 수행을 10 회 반복한 후, 이 중 최대, 최소값을 제외한 나머지의 평균값을 취하는 방식으로 측정하였다. 경우 I 은 SOP 모듈을 통해 legacy TCP/IP 스택을 사용하는 경우와 기존 소켓 계층을 거쳐 legacy TCP/IP 스택을 사용하는 경우에 대한 send 시스템 호출의 지연시간 비교이다. 이와 유사하게, 경우 II는 SOP 모듈을 거치는 경우와 그렇지 않은 경우의 오프로드 TCP/IP 스택에서의 send 시스템 호출에 해당한다. 경우 I 에서는 SOP 모듈을 사용하는 편이 SOP 모듈을 거치지 않는 경우보다 약 4.8 % 정도 지연시간이 증가하였다. 한편 오프로드 TCP/IP를 통한 send 시

표 1. SOP 모듈 사용 시의 send 시스템 호출의 성능 (단위: us)

	Legacy TCP/IP (경우 I)		Offload TCP/IP (경우 II)	
	No SOP	SOP	No SOP	SOP
send 시스템 호출의 지연시간 (latency)	29.4	30.8	32.3	33.7

스템 호출의 경우 (경우II)에는 SOP 모듈을 사용하는 경우, SOP 모듈을 거치지 않는 경우보다 약 4.3% 정도 지연시간이 증가하였다. 표 1를 통해 SOP 모듈이 제공하는 새로운 기능의 사용으로 인해 개별 시스템 호출에 약간의 성능 저하가 발생하고 있음을 알 수 있다.

다음은 SOP 모듈이 네트워크 서버 응용의 성능에 어떤 영향을 미치는지 분석하기 위해서 대표적인 네트워크 서버인 웹 서버에 대한 성능 평가를 수행하였다. 실험에 사용된 웹 서버는 HTTP/1.0 프로토콜을 지원하는 자체 제작한 간단한 웹 서버이며, 성능 측정에 필요한 클라이언트 워크로드를 발생시키기 위해서 대표적인 웹 서버 벤치마크인 SpecWeb99를 사용하였다. 기본적인 성능 측정은 웹 서버가 SOP 모듈을 통해 오프로드 TCP/IP 스택을 사용하여 웹 서비스를 수행하는 경우에 대해 수행하였다. 또한 SOP 모듈이 성능에 미치는 영향을 비교하기 위해 웹 서버가 기존의 리눅스 TCP/IP를 그대로 사용하는 경우와 (SOP 모듈 없이) 오프로드 TCP/IP를 사용하는 경우에 대한 측정도 함께 수행하였다.

그림 5는 SpecWeb99이 발생시키는 부하(단위 시간 당 웹 서비스 요청)에 대한 웹 서버의 처리율(throughput)을 나타내고 있다. 이 그림에서 leg_tcpip는 기존 리눅스 커널의 소켓 계층과 TCP/IP 프로토콜 스택을 그대로 사용하는 웹 서버의 경우이고, offload_no_sop는 SOP 모듈 없이 직접 TOE의 오프로드 TCP/IP 스택을 사용하는 경우를, offload_sop는 SOP 모듈을 통해 오프로드 TCP/IP 스택을 사용하는 경우를 나타낸다. 웹 서비스 요청률이 낮을 때에는, 세 경우 모두 요구된 웹 서비스 요청을 무리없이 처리하고 있다. 그러나 웹 서비스 요청률이 점차 높아짐에 따라 기존 리눅스 네트워킹을 그대로 사용하는 웹 서버의(leg_tcp) 경우 요청률이 약 700 requests/sec인 시점에서 포화 상태에 이르러, 더 이상의 처리율의 증가가 나타나지 않는다. 이에 비해 오프로드 TCP/IP를 사용하는 offload_sop와 offload_

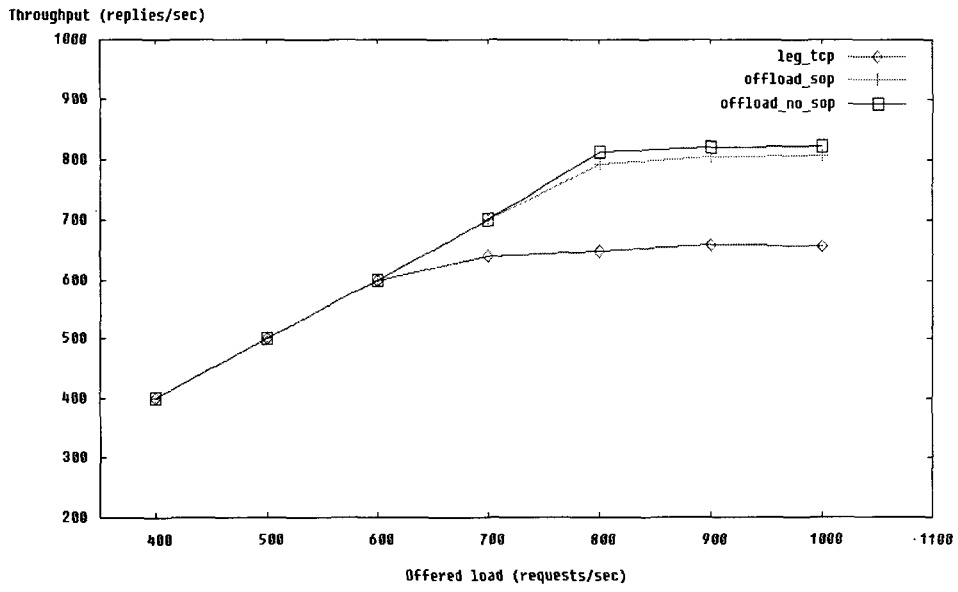

그림 5. 웹 서버의 처리율

no_sop 의 경우는 leg_tcp 경우에 비해 처리율 측면에서 각각 약 26%와 28% 정도의 성능 향상을 보이고 있다(이와 같은 성능 향상은 물론 TOE 사용에 기인한 것이다). 여기서 주목할 점은 SOP 모듈을 통해 오프로드 TCP/IP를 사용하는 웹 서버의 성능이 SOP 모듈을 사용하지 않는 경우에 비해 처리율 증가가 약 2~3% 정도 작다는 점이다. 이 차이는 offload_sop 경우에서 SOP 모듈의 프로토콜 선택 기능이 동작한 영향으로 볼 수 있다.

그림 6은 앞서 처리율에 대한 실험 중 leg_tcp 경우가 포화 상태에 이르는 시점에서 세 가지 경우에 대한 호스트 측의 CPU 이용률(utilization)을 측정한 결과이다. 이 시점에서 leg_tcp 경우는 포화 상태에 이르러 100%의 CPU 이용률을 보이는 반면, offload_sop와 offload_no_sop의 CPU 이용률은 각각 64% 와 61% 정도이다. 특히 두 경우의 사용자 모드 이용률은 약 19% 정도로 비슷하였으나, 시스템(또는 커널) 모드에서의 이용률은 각각 45%와 42%로 차이를 보이고 있다. 즉, 앞서 처리율 실험의 경우와 유사하게, 단순히 오프로드 TCP/IP를 사용하는 경우보다 SOP 모듈을 통해 오프로드 TCP/IP를 사용하는 경우 (특히 시스템 모드에서의) CPU 이용률의 증가가 나타나고 있으며, 이 또한 SOP 모듈의 프로토콜 선택 기능의 동작으로 인한 영향으로 볼 수 있다.

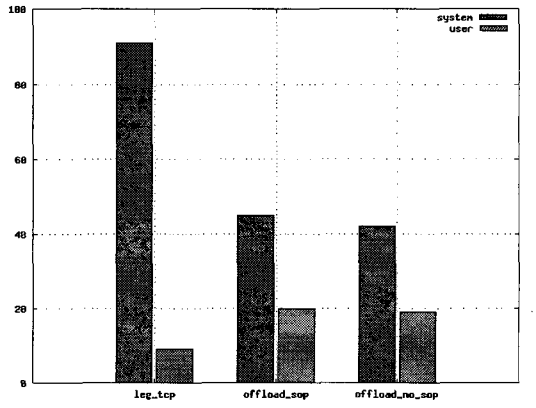

그림 5. 웹 서버의 CPU 이용률

5. 결 론

SOP 계층은 TOE를 위한 소켓 계층으로, 소켓을 사용하는 네트워크 응용 프로그램이 일반적인 NIC 카드를 사용하는 것과 동일한 방식으로 TOE를 사용할 수 있도록 해 준다. 기본적으로 소켓 계층은 표준 BSD 소켓 인터페이스를 통해 TOE의 오프로드 TCP/IP 프로토콜 스택을 사용할 수 있게 해 준다. 특히 기존 소켓 응용 프로그램에 대해 바이너리 수준의 호환성을 제공함으로써, 기존 소켓 응용 프로그램들이 수정 없이 TOE 상에서 동작이 가능하다. SOP 소켓 계층의 구체적인 특징은 다음과 같다; (1) TOE 사용에 있어서 기존의 모든 소켓 관련 API 와 (소켓에 대한) 파일 입출력 API 를 그대로 사용할 수 있다. (2) TOE 카드 사용 시 기존 소켓 프로그램 실행에 대한 바이너리 수준의 호환성이 제공된다. (3) 여러 TOE 카드와 NIC 을 한 시스템에서 동시에 사용 가능하다. (4) 여러 유형의 TOE 구현에 대해 확장이 용이하다.

참 고 문 헌

[1] M. Rangarajan, A. Bohra, K. Banerjee, E.V. Carrera, R. Bianchini, L. Iftode, and W. Zwaenepoel, "TCP Servers: Offloading TCP Processing in Internet Servers. Design, Implementation, and Performance," *Tech Report of Rutgers University*, 2002.

[2] Adaptec, ASA 7211 and ANA 7711, http://www.adaptec.com.

[3] Alacritech, Storage and Network Acceleration, http://www.alacritech.com.

[4] Windriver, Tornado for Intelligent Network Acceleration, http://www.windriver.com.

[5] J.M. Smith and C.B.S. Traw, "Giving Applications Access to Gb/s Networking," *IEEE Network*, Vol. 7, No. 4, pp. 44-52, 1993.

[6] M.K. McKusick, K. Bostic, M.J. Karels, and J.S. Quarterman, *The Design and Implementationf of the 4.4 BSD Operating System*, Addison Wesley, 1994.

[7] G.R. Wight and W.R. Stevens, *TCP/IP Illustrated, Volume2: The Implementation*, Addison Wesley, 2001.

[8] J. Pinkerton, "SDP: Sockets Direct Protocol," in *Proc. of Infiniband developers Conference*,

2001.

[9] D.P. Bovet and M. Cesati, *Understanding the Linux Kernel 2nd Ed.*, Oreilly, 2003.

손 성 훈

1991년 서울대학교 계산통계학과 졸업(학사)
1993년 서울대학교 대학원 전산과학과(석사)
1999년 서울대학교 대학원 전산과학과(박사)
1999년~2004년 한국전자통신연구원 선임연구원
2004년~현재 상명대학교 소프트웨어학부 전임강사
관심분야 : 멀티미디어, 시스템소프트웨어, 임베디드시스템