10장. 네트워크 (Networks)

네트워킹과 리눅스는 거의 동의어이다. 리눅스는 말 그대로 인터넷 또는 월드 와이드 웹 (World Wide Web, WWW)의 산물이다. 리눅스의 개발자와 사용자들은 정보와 프로그램 코드 을 교환하기 위해 웹을 사용하며, 조직의 네트워킹 요구를 처리하기 위해 리눅스를 자주 사 용한다[1]. 이 장은 리눅스가 통틀어 TCP/IP라고 부르는 네트워크 프로토콜을 어떻게 지원하 는지 설명한다.

TCP/IP는 미국 정부가 출자하는 미국 연구망(ARPANET)에 연결된 컴퓨터 간의 통신을 지원 하기 위해 구상된 것이다. ARPANET은 패킷 스위칭과 하나의 프로토콜이 다른 프로토콜의 서비스을 사용하는 프로토콜 계층화 등의 네트워킹 개념을 창시했다. ARPANET은 1988년에 종료되었지만 그 계승자인 NSF[2] NET과 인터넷은 더 크게 성장했다. 현재 월드 와이드 웹이 라고 알려진 것은 ARPANET으로부터 성장했으며, TCP/IP 프로토콜을 바탕으로 하고 있다. ARPANET 상에서는 유닉스가 광범위하게 사용되었으며, 처음으로 네트워킹이 가능한 유닉 스 버전은 4.3 BSD였다. 리눅스의 네트워킹 구현은 4.3 BSD를 모델로 설계되었으며, 리눅스 는 (약간 확장된) BSD 소켓과 TCP/IP 네트워킹 전체를 지원한다. 리눅스에서 이 TCP/IP 프 로그래밍 인터페이스를 선택한 이유는 TCP/IP가 널리 사용되고 있으며, 리눅스와 다른 유닉 스 플랫폼과의 응용 프로그램 호환성을 높이기 위한 것이었다.

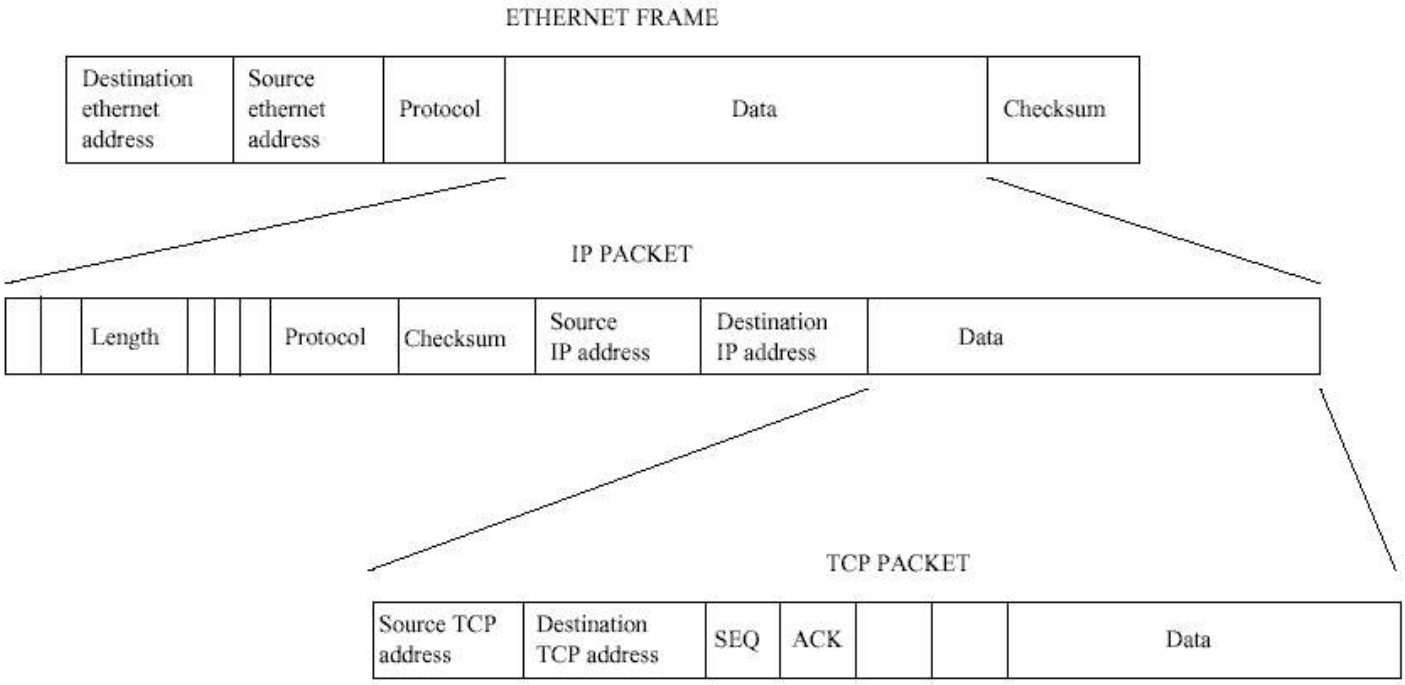

그림 10.1 : TCP/IP 프로토콜 계층

10.1 TCP/IP 네트워킹의 개관

이 절은 TCP/IP 네트워킹의 기본 원리에 대한 개관이다. 이것은 (이후의 절과 같은) 상세한 설명이 아니기 때문에 한번 읽어보기 바란다.

IP 네트워크에서는 각 기계를 고유하게 식별하는 32비트 숫자인 IP 주소를 각 기계에 부여 한다. WWW는 매우 거대하고 계속 성장하는 IP 네트워크로서, WWW에 연결된 모든 기계 들은 할당된 고유한 IP 주소를 가진다. IP 주소는 예를 들어 16.42.0.9와 같이 점으로 구분되 는 네 개의 숫자로 나타낸다. 실제로는 네트워크 주소와 호스트 주소의 두 부분으로 IP 주 소를 구분한다. (IP 주소에는 여러 클래스들이 있어서) 각 부분의 크기는 달라질 수 있지만, 16.42.0.9를 예로 들면 16.42는 네트워크 주소이고 0.9는 호스트 주소가 된다. 호스트 주소 는 서브네트워크와 호스트 주소로 더 (자세히) 나눌 수 있다. 다시 16.42.0.9를 예로 들면, 서브네트워크 주소는 16.42.0이 되고 호스트 주소는 16.42.0.9가 된다. 이렇게 IP 주소를 몇 구획으로 나눌 수 있으므로, (네트워크를 사용하는) 기관은 자신의 네트워크를 몇 구획으로 나눌 수 있다. 예를 들어 16.42가 ACME 컴퓨터사의 네트워크 주소라면, 16.42.0는 서브네트 워크 0번, 16.42.1은 서브네트워크 1번이 될 것이다. 이 서브네트워크는 서로 다른 건물에 있을 수도 있고, 임대 전화선을 이용하거나 무선(통신수단)을 이용해 연결되어 있을 수도 있 다. IP 주소는 네트워크 관리자가 할당하는데, IP 서브네트워크를 사용하여 네트워크 관리 부 담을 분산시킬 수 있다. IP 서브네트워크 관리자는 자신의 IP 서브네트워크 안에서 자유롭게 IP 주소를 할당할 수 있다.

하지만 일반적으로 IP 주소는 아주 기억하기 어렵다. 이름을 붙이는 것이 훨씬 (기억하기) 쉽다. linux.acme.com이 16.42.0.9보다 훨씬 더 기억하기 쉬운데, (이름을 사용하기 위해서 는) 네트워크 이름을 IP 주소로 변환해 주는 도구가 필요하다. 이 이름들을 /etc/hosts 파 일에 정적으로 명시할 수도 있지만, 리눅스는 분산 네임 서버(Distributed Name Server, DNS)에 이 이름들을 변환해 달라고 요청할 수도 있다. 이 경우 로컬 호스트는 하나 이상의 DNS 서 버의 IP 주소를 알고 있어야만 하는데, 이 주소들을 /etc/resolv.conf에 기록한다.

웹 페이지를 읽을 때와 같이 다른 기계에 접속할 때마다 그 기계와 자료를 교환하기 위해 그 기계의 IP 주소를 사용한다. 자료들은 IP 패킷에 담겨 전달되는데, 각 패킷마다 출발지 기계와 목적지 기계의 IP 주소, 체크섬(checksum) 및 다른 유용한 정보를 담고 있는 IP 헤더 가 붙어 있다. 체크섬은 IP 패킷에 있는 데이터를 가지고 계산하는데, 이를 이용하여 IP 패킷 수신자는 전화선의 잡음 등으로 인해 전달과정에서 패킷이 손상되었는지를 판단할 수 있 다. 응용 프로그램이 보내는 데이터는 좀 더 다루기 쉬운 작은 패킷들로 쪼개질 수 있다. IP 데이터 패킷의 크기는 연결 매체에 따라 달라지는데, 일반적으로 이더넷 패킷이 PPP 패킷보 다 더 크다. 목적지 호스트는 데이터 패킷들을 다시 조합하여 응용 프로그램에 데이터를 건 내준다. 느린 시리얼 링크를 통해 많은 그래픽 이미지들을 담고 있는 웹 페이지를 보면 위 에서 말한 데이터의 분해와 조립 과정을 그림을 보듯 살펴볼 수 있다.

같은 IP 서브네트워크에 연결되어 있는 호스트끼리는 IP 패킷을 직접 보낼 수 있지만, 그렇 지 않은 경우에는 게이트웨이(gateway)라고 하는 특별한 호스트에 IP 패킷을 보내야만 한다. 게이트웨이(또는 라우터)는 하나 이상의 IP 서브네트워크에 연결되어 있는데, 한 IP 서브네 트워크에서 받은 패킷을 다른 IP 서브넷으로 전송한다. 예를 들어, 서브네트워크 16.42.1.0과 16.42.0.0이 어떤 게이트웨이를 통해 연결되어 있다면 서브네트워크 0에서 서브네트워크 1 로 전달되는 패킷들은 게이트웨이로 보내지고 게이트웨이는 이 패킷을 전달한다. 각 호스트 들은 정확한 기계에 IP 패킷을 전달하기 위해 라우팅 테이블(routing table)을 작성한다. 라우 팅 테이블에는 모든 IP 목적지에 대해 그 목적지에 도달하기 위해 어떤 호스트에 IP 패킷을 전달해야 하는지를 결정하기 위해 사용되는 정보가 있다. 이 라우팅 테이블은 동적이어서 응용 프로그램이 네트워크를 사용하거나 네트워크 구성도가 변경되거나 하면 시간이 지남에 따라 변경된다.

IP 프로토콜은 다른 프로토콜이 데이터를 보낼 때 사용하는 전송 계층이다. TCP는 신뢰할 수 있는 일대일 프로토콜로서, 데이터를 주고 받기 위해 IP 프로토콜을 사용한다. IP 패킷에 헤더가 붙어 있는 것처럼, TCP 패킷에도 헤더가 붙어 있다. TCP는 연결 중심적인 프로토콜 로 (이를 사용하는) 두 네트워크 응용 프로그램은 그 사이에 많은 서브네트워크, 게이트웨이 및 라우터가 있더라도 단일한 가상의 접속을 통해 연결된다. TCP는 두 응용프로그램간의 데이터를 신뢰할 수 있는 방식으로 전달하며 데이터의 손실이나 중복이 없다는 것을 보장한 다. TCP가 IP를 사용하여 TCP 패킷을 전송할 때, IP 패킷에 들어있는 데이터는 바로 TCP 패킷이다. 서로 통신하고 있는 호스트의 IP 계층은 IP 패킷을 주고 받는 역할을 한다. UDP 도 (UDP) 패킷을 전달하는데 IP 계층을 사용하지만, TCP와는 달리 UDP는 신뢰할 수 없는 프로토콜이며 데이터그램(datagram) 서비스를 제공한다. 이와 같이 다른 프로토콜이 IP를 사 용하려면, IP 패킷을 받을 때 IP 계층이 이 IP 패킷에 담긴 데이터를 어떤 상위 프로토콜에 전달해야 하는지를 알고 있어야만 한다. 이를 위해 모든 IP 패킷 헤더에는 프로토콜 식별자 를 지정하는 바이트가 있다. TCP가 IP 계층에다 IP 패킷을 전송하도록 요청하면, 그 패킷에 TCP 패킷이 들어있다는 것을 IP 패킷 헤더에 기록한다. IP 계층이 데이터를 받으면, 이 프로 토콜 식별자를 사용하여, 받은 데이터를 상위의 어떤 계층에 전달할지를 결정한다. 이 경우 에는 TCP 계층이 될 것이다. 응용프로그램이 TCP/IP를 통해 통신을 할 때, 응용프로그램은 상대방의 IP 주소뿐만 아니라 그 응용프로그램의 포트 주소 또한 명시하여야 한다. 포트 번 호는 응용프로그램마다 유일하며, 표준 네트워크 응용프로그램은 표준 포트번호를 사용한다. 예를 들어, 웹서버는 80번 포트를 사용한다. 이러한 등록된 포트번호는 /etc/services에 서 볼 수 있다.

프로토콜의 계층구조는 TCP, UDP 및 IP로 (구분하는 것으로) 끝나는 것이 아니다. IP 프로토 콜 자체도 IP 패킷을 다른 IP 호스트로 전송하는데 수많은 장치들을 사용한다. 이 장치는 자신만의 프로토콜 헤더를 추가하기도 한다. 이러한 예로는 이더넷 계층이 있으며, 또 다른 예로 PPP와 SLIP이 있다. 이더넷 네트워크에서 많은 호스트가 실제 케이블 하나에 동시에 접속할 수 있다. 전송되는 모든 이더넷 프레임은 연결된 모든 호스트에 보이게 되므로[3] 모 든 이더넷 장치는 고유한 주소

를 갖는다. 호스트는 자기 주소로 배달되는 모든 이더넷 프레 임을 받아들이지만, 같은 네트워크에 연결된 다른 호스트들은 이를 무시하게 된다. 이더넷의 이런 유일한 주소는 이더넷 장치를 만들 때 적어넣게 되는데, 일반적으로 이더넷 카드의 SROM[4]에 들어 있다. 이더넷 주소는 6바이트 길이인데 예를 들면 08-00-2B-00-49-A4같은 값 을 갖는다. 어떤 이더넷 주소는 멀티캐스트(multicast) 목적으로 예약되어 있는데, 이런 주소 로 보내지는 이더넷 프레임은 같은 네트워크 안에 있는 모든 호스트가 받는다. 이더넷 프레 임은 (데이터로) 수많은 프로토콜들을 전송할 수 있기 때문에, IP 패킷과 같이 헤더에 프로 토콜 식별자가 있다. 이에 따라 이더넷 계층은 정확하게 IP 패킷을 받아 IP 계층에 전달할 수 있다.

이더넷과 같은 다중 접속 프로토콜을 통해 IP 패킷을 보내기 위해서는 IP 계층은 IP 호스트 의 이더넷 주소를 찾아야만 한다. IP 어드레스는 단지 개념적인 주소일 뿐이고, 고유한 물리 적인 주소를 가지고 있는 것은 이더넷 장치이기 때문이다. 반면에 IP 주소는 네트워크 관리 자의 의지대로 지정되고 변경될 수 있지만, 네트워크 하드웨어는 자신의 물리적 주소 또는 모든 기계가 받아야만 하는 특별한 멀티캐스트에만 반응한다. 리눅스는 IP 주소를 이더넷 주소와 같은 실제 하드웨어 주소 변환하기 위해 ARP(Address Resolution Protocol)를 사용한다. 특정한 IP 주소를 가진 하드웨어 주소를 알고자 하는 호스트는 변환하고자 하는 IP 주소가 담긴 ARP 요청 패킷을 멀티캐스트 주소에 보내 모든 노드에 전달한다. 그 IP 주소를 가지 고 있는 호스트는 자신의 하드웨어 주소가 담긴 ARP 응답을 돌려준다. ARP는 이더넷 장치 만 사용되는 것이 아니라 IP 주소를 FDDI와 같은 다른 물리적 장치의 주소로 변화하는데에 도 사용할 수 있다. ARP를 할 수 없는 네트워크 장치들은 따로 표시를 해 두어 리눅스는 (이 장치에 대해서는) ARP를 시도하지 않는다. 이와는 반대되는 기능으로 RARP(Reverse Address Resolution Protocol)가 있는데, 이것은 물리적 네트워크 주소를 IP 주소로 변환한다. 이 기능은 게이트웨이가 사용하는데, 게이트웨이는 원격 네트워크에 있는 IP 주소를 대신해 서 ARP 요청에 응답한다.

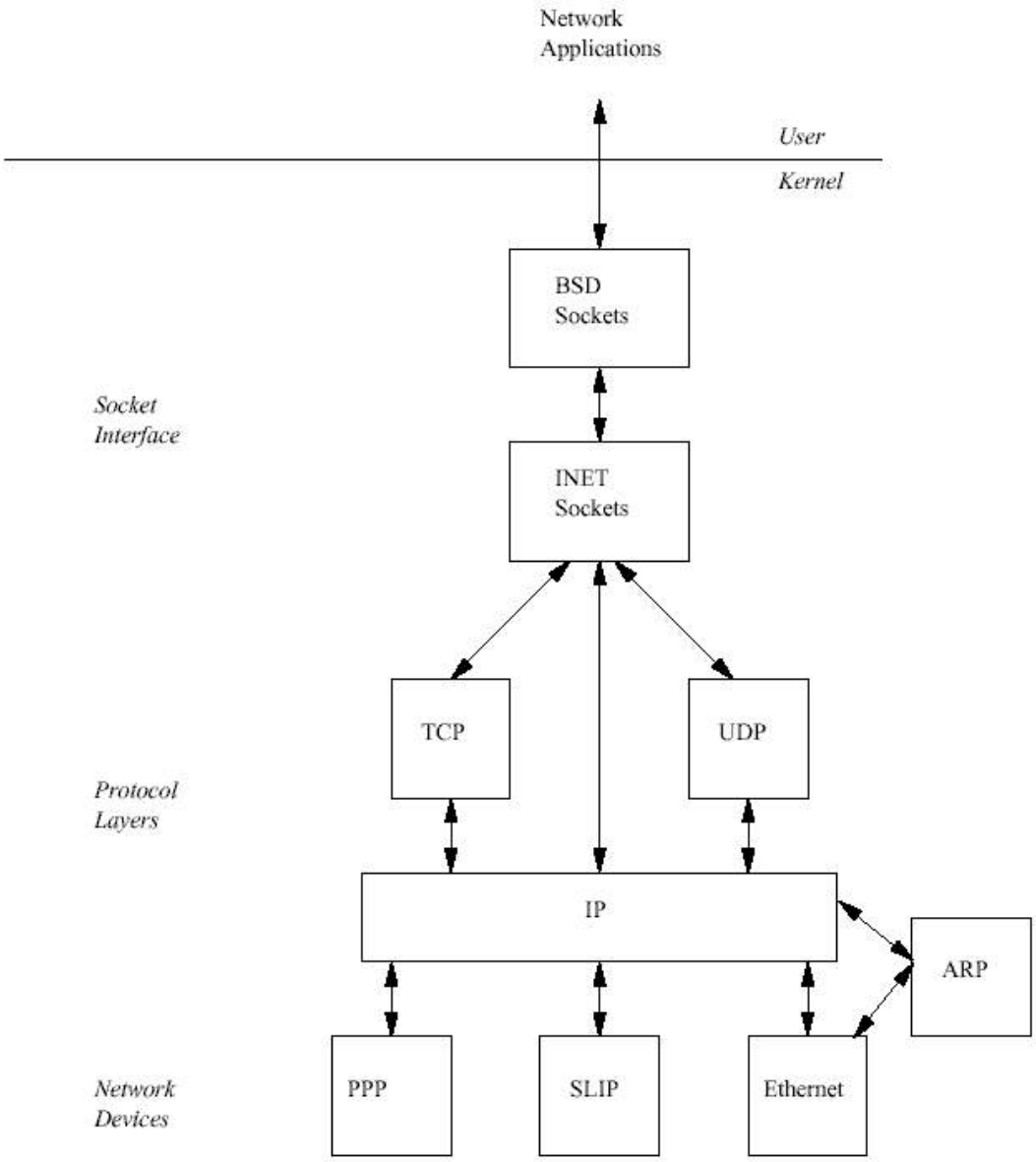

그림 10.2 : 리눅스의 네트워킹 계층

10.2 리눅스의 TCP/IP 네트워킹 계층

네트워크 프로토콜과 마찬가지로, 그림 10.2에서 볼 수 있는 것처럼 리눅스는 인터넷 프로 토콜 주소 패밀리(address family)를 일련의 연관된 소프트웨어 계층으로 구현하고 있다. BSD 소켓은 BSD 소켓만 처리하는 일반적인 소켓 관리 소프트웨어가 지원한다. INET 소켓 계층 은 소켓 관리 소프트웨어를 지원하는데, 이것은 IP 기반의 프로토콜인 TCP와 UDP의 통신 종점을 관리한다. UDP(User Datagram Protocol)는 비연결지향 방식의 프로토콜(connectionless protocol)인데 비해, TCP(Transmission Control Protocol)는 연결지향의 신뢰할 수 있는 일대일 프 로토콜이다. UDP 패킷을 전송할 때, 리눅스는 그 패킷이 목적지에 안전하게 도착하였는지를 알 수도 없고 신경을 쓰지도 않는다. TCP 패킷들에는 번호를 매겨, TCP 접속의 양 끝(종점 호스트)은 전송 데이터가 정확하게 수신되었는지를 확인한다. IP 계층에는 인터넷 프로토콜을 구현한 코드가 들어 있다. 이 코드는 전송하는 데이터 앞에 IP 헤더를 붙이고, 들어오는 IP 패킷을 TCP나 UDP 계층으로 어떻게 전달하는지를 알고 있다. IP 계층 아래에서 PPP 또 는 이더넷과 같은 네트워크 장치들이 리눅스의 모든 네트워킹을 지원한다. 네트워크 장치라 고 항상 물리적인 장치만을 가리키는 것은 아니다. 루프백 장치와 같은 몇몇 장치는 순전히 소프트웨어로만 작성되어 있다. mknod 명령으로 만들어지는 표준적인 리눅스 장치와는 달 리, 네트워크 장치는 관련된 소프트웨어가 장치를 찾아내 초기화해야지만 나타난다. 그래서 해당하는 이더넷 디바이스 드라이버를 넣어서 커널을 빌드해야만 /dev/eth0를 볼 수 있 다. ARP 프로토콜은 IP 계층과 각종 주소에 대한 ARP를 지원하는 프로토콜 사이에 있다.

10.3 BSD 소켓 인터페이스(Socket Interface)

BSD 소켓 인터페이스는 다양한 형태의 네트워킹 뿐만 아니라 프로세스간 통신도 지원하는 일반적인 인터페이스이다. 소켓은 통신 연결의 한쪽 끝으로 생각할 수 있는데, 통신한고 있 는 두 프로세스는 통신 연결에서 자신쪽 끝에 해당하는 소켓을 가지게 된다. 소켓을 특별한 종류의 파이프로 생각할 수도 있지만, 파이프와는 달리 소켓은 거기에 담을 수 있는 데이터 의 양에 제한이 없다. 리눅스는 몇 가지 클래스의 소켓을 지원하는데, 이것들을 주소 패밀리 (address family)라고 부른다. 이는 각 클래스별로 자신의 통신에 사용하는 주소 표현법을 가 지고 있기 때문이다. 리눅스는 다음과 같은 소켓 주소 패밀리 또는 도메인을 지원한다.

```
UNIX            유닉스 도메인 소켓 (Unix domain socket)
INET            TCP/IP 프로토콜을 이용한 통신을 지원하는 인터넷 주소 패밀리
AX25            아마추어 라디오 X.25
IPX             노벨의 IPX 프로토콜
APPLETALK       애플사의 Appletalk DDP 프로토콜
X25             X.25 프로토콜
```

소켓에는 몇가지 타입이 있으며, 이는 접속을 지원하는 서비스의 종류를 나타낸다. 모든 주 소 패밀리가 모든 형태의 서비스를 지원하는 것은 아니다. 리눅스 BSD 소켓은 몇가지 소켓 타입을 지원한다.

- **스트림(Stream)** 이 소켓은 데이터가 전송 중 분실, 오염 또는 중복되지 않는다는 것을 보장 하는 신뢰할 수 있는 양방향 순차 데이터 스트림을 제공한다. INET 주소 패밀리의 TCP 프로토콜이 스트림 소켓을 지원한다.
- **데이터그램(Datagram)** 이 소켓은 양방향 데이터 전송을 제공하지만, 스트림 소켓과는 달리 그 메시지가 (제대로) 도착한다는 것을 보장하지는 않는다. 메시지가 목적지에 도착하였 다 하더라도, 메시지가 순서에 맞게 또는 중복되거나 오염되지 않고 도착하였다는 것을 보장하지 않는다. INET 주소 패밀리의 UDP 프로토콜이 이 종류의 소켓을 지원한다.
- **가공하지 않은(Raw)** 프로세스가 하부 프토토콜에 직접 접근(그래서 "raw")할 수 있는 소켓 이다. 예를 들면 이더넷 장치에 이 소켓을 열어 가공되지 않은 IP 데이터 흐름을 지켜보 는 것이 가능하다.
- **도착 신뢰 메시지(Reliable Delivered Messages)** 이것은 데이터그램 소켓과 아주 비슷하지만 데이터가 (목적지에) 도착한다는 것을 보장한다.
- **순차적 패킷(Sequenced Packets)** 이것은 스트림 소켓과 비슷한데 데이터 패킷의 크기가 고정 되어 있다.
- **패킷(Packet)** 이것은 표준 BSD 소켓 타입은 아니고, 장치 수준에서 프로세스가 직접 패킷에 접근할 수 있는 리눅스 특유의 확장이다.

소켓을 사용하여 통신을 하는 프로세스는 클라이언트 서버 모델을 따른다. 서버는 서비스를 제공하고 클라이언트는 이 서비스를 이용한다. 이런 예로 웹 페이지를 제공하는 웹 서버와 그 페이지들을 읽는 웹 클라이언트 또는 브라우저를 들 수 있다. 소켓을 사용하는 서버는 먼저 소켓을 만든 후 소켓에 이름을 바인드(bind)한다. 이 이름의 형식은 소켓의 주소 패밀 리에 따라 달라지는데, 실제로는 서버의 로컬 주소가 된다. 소켓의 이름 또는 주소는 sockaddr 자료 구조를 이용해 명시한다. INET 소켓은 그것에 바인드된 IP 포트 주소를 가 지게 된다. 등록된 포트 번호는 /etc/services에서 볼 수 있다. 예를 들어, 웹 서버의 포 트번호는 80번이다. 소켓에 주소가 바인드되었다면, 서버는 그 바인드된 주소를 가리키는 연결 요청이 들어오는지 리슨(listen)을 한다. 연결 요청을 하는 클라이언트는 소켓을 만들고 서버의 주소를 명시하여 소켓에 대해 연결 요청을 한다. INET 소켓에서 서버의 주소는 서버 의 IP 주소와 포트 번호이다. 이러한 연결 요청은 다양한 프로토콜 계층을 통해 전달되어 서버의 리슨 소켓에 도달하게 된다. 서버가 연결 요청을 받으면, 이것을 받아들이거나 (accept) 또는 거부한다(reject). 연결 요청을 받아들이기로 하였다면, 서버는 연결을 받아들일 새로운 소켓을 만든다. 연결 요청을 리슨하는데 사용하는 소켓은 연결을 받아들이는데 사용 할 수는 없다. 연결이 이루어지고 나면, 서버와 클라이언트는 자유롭게 데이터를 주고 받을 수 있다. 마지막으로, 연결이 더이상 필요없는 경우 소켓을 종료(shutdown)할 수 있다. 이 때 전송 중에 있는 데이터 패킷이 정확하게 처리되었는지에 유의하여야 한다.

BSD 소켓에 어떤 조작을 가하는 것이 무엇을 의미하는지는 어떤 주소 패밀리 위에서 작업 을 하고 있느냐에 따라 다르다. TCP/IP 접속을 설정하는 것은 아마추어 라디오 X.25 접속을 설정하는 것과는 아주 다르다. 가상 파일 시스템과 마찬가지로 리눅스는 BSD 소켓 계층으 로 소켓 인터페이스를 추상화한다. BSD 소켓 계층은 BSD 소켓 계층이 응용프로그램과 인터 페이스하는 것에 관련된다. 이런 소켓 인터페이스는 독립된 주소 패밀리를 가지는 소프트웨 어에 의해 지원을 받는다. 커널 초기화 과정에서, 커널에 구현된 주소 패밀리는 (자신이 지 원하는) BSD 소켓 인터페이스와 함께 자신을 등록한다. 나중에 응용프로그램이 BSD 소켓을 만들고 사용할 때, BSD 소켓과 그것이 지원하는 주소 패밀리 사이의 연관이 만들어진다. 이 러한 연관관계는 교차연결 자료구조와 주소 패밀리 고유의 지원 루틴 테이블을 통해 만들어 진다. 예를 들어 응용프로그램이 새로운 소켓을 만들 때 BSD 소켓 인터페이스가 사용하는 주소 패밀리 고유의 소켓 생성 루틴이 있다.

커널을 설정할 때 (많은) 주소 패밀리와 프로토콜을 protocols 벡터에 넣는다. protocols 벡터에는 각 주소 패밀리 또는 프로토콜의 이름 (예를 들면 "INET")과 초기화 루틴이 들어 간다. 시스템이 부팅되면서 소켓 인터페이스를 초기화할 때, 각 프로토콜의 초기화 루틴이 불리게 된다. 여기서 소켓 주소 패밀리 별로 일련의 프로토콜 연산 루틴을 등록하게 된다. 이것은 루틴들의 집합이며 각 루틴은 해당 주소 패밀리의 고유한 특정 연산을 수행한다. proto_ops 자료구조는 주소 패밀리 타입과 특정 주소 패밀리에 고유한 소켓 연산 루틴에 대한 포인터들의 집합으로 이루어져 있다. pops 벡터는 인터넷 주소 패밀리같은 (AF_INET 은 2이다) 주소 패밀리 식별자로 인덱스 되어있다.

10.4 INET 소켓 계층

INET 소켓 계층은 TCP/IP 프로토콜들을 포함하는 인터넷 주소 패밀리를 지원한다. 위에서 설명한 것처럼 이들 프로토콜들은 계층적이고, 한 프로토콜이 다른 프로토콜의 서비스를 사 용한다. 리눅스의 TCP/IP 코드와 자료구조는 이 계층구조를 반영한다. BSD 소켓 계층으로의 인터페이스는 네트워크 초기화 도중에 BSD 소켓 계층에 등록 한 인터넷 주소 패밀리 소켓 함수들을 통한다. 이들은 등록된 다른 주소 패밀리와 함께pops 벡터에서 보관한다. BSD 소 켓계층은 등록된 INET proto_ops 자료구조로부터 INET 계층의 소켓지원 루틴을 호출하여 필요한 일을 수행한다. 예를 들어, 주소 패밀리에 INET을 주고 BSD 소켓을 만들라고 요구 한다면, 이는 밑에 있는 INET 소켓 생성 함수를 사용하게 된다. BSD 소켓 계층은 이들 각각 의 함수마다 INET 계층에 BSD 소켓을 나타내는 socket 자료구조를 전달한다. BSD socket을 TCP/IP에만 필요한 정보로 어지럽히기 보다는 INET 소켓 계층은 자신만의 자료 구조인 sock을 가지고 자신을 BSD socket 자료구조와 연결한다. 이런 연결은 그림 10.3 에서 볼 수 있다. sock 자료구조는 BSD socket에 있는 data 포인터를 통해 BSD socket 자료구조와 연결된다. 이것은 계속된 INET 소켓 호출에서 쉽게 sock 자료구조를 얻어올 수 있다는 의미이다. sock 자료구조의 프로토콜 함수 포인터 역시 생성시에 셋업이 되며, 이는 요구한 프로토콜에 따라 다르다. 만약 TCP를 요구했다면, sock 자료구조의 프로토콜 함수 포인터는 TCP 연결을 위해 필요한 TCP 프로토콜 함수 집합을 가리킬 것이다.

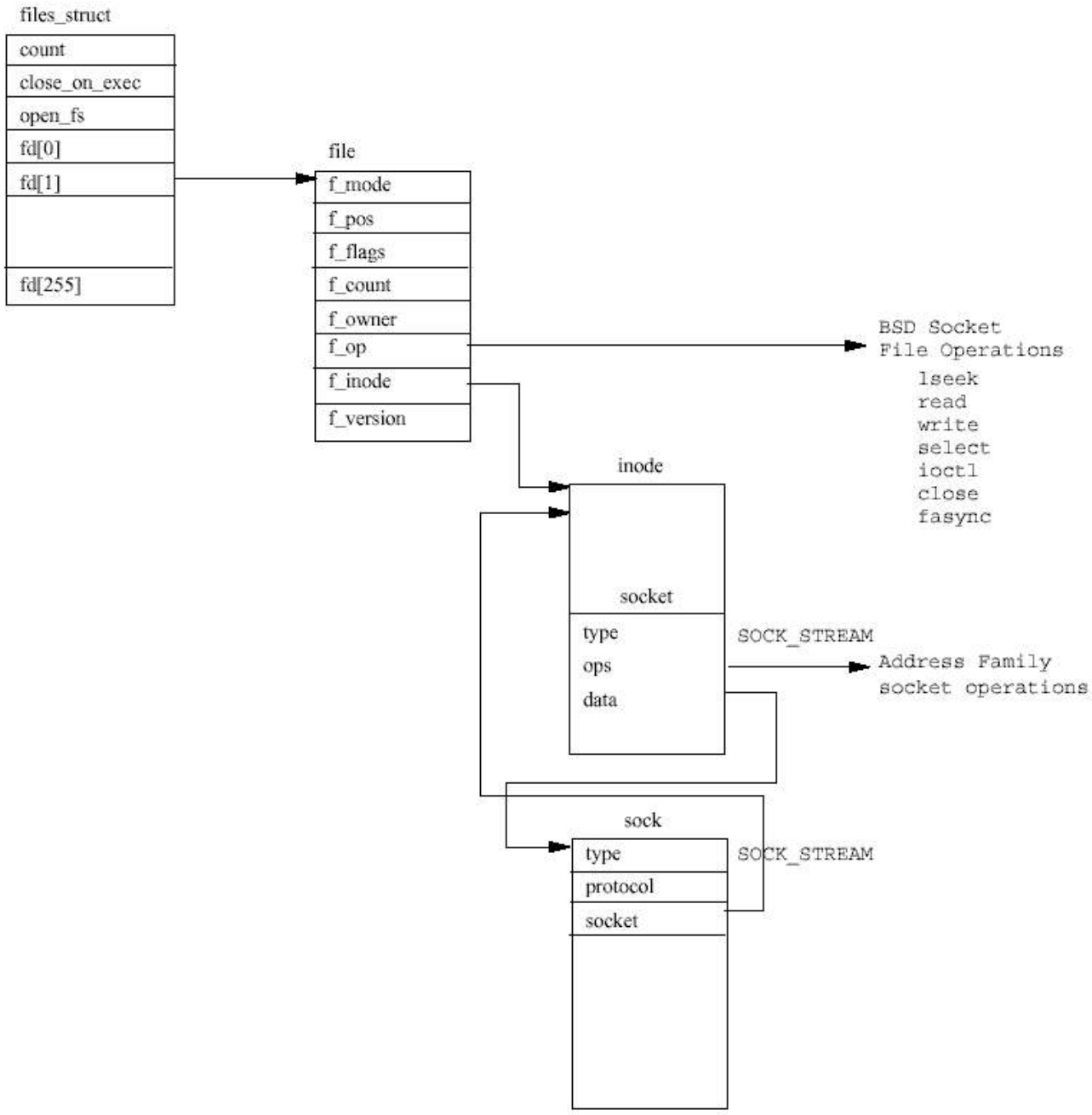

그림 10.3 : 리눅스 BSD 소켓 자료구조

10.4.1 BSD 소켓 만들기

새 소켓을 만드는 시스템 콜에는 주소 패밀리 식별자와 소켓 타입, 그리고 프로토콜을 인자 로 준다. 먼저, 요구한 주소 패밀리를 사용하여 pops 벡터에서 일치하는 주소 패밀리가 있 는지 찾는다. 어떤 주소 패밀리는 커널 모듈로 만들어져 있을 수도 있는데, 이 경우 kerneld 데몬이 이 모듈을 읽어들여야 작업을 계속할 수 있다. BSD 소켓을 나타내기 위해 새 socket 자료구조를 할당한다. 실질적으로 socket 자료구조는 물리적으로 VFS inode 자료구조의 한 부분이고 소켓을 할당한다는 것은 실제로는 VFS inode를 할당한다는 것을 의미한다. 이는 소켓이 일반 파일과 똑같은 방법으로 작동한다는 것을 생각한다면 별로 이 상하게 보이진 않을 것이다. 모든 파일은 VFS inode 자료구조로 나타내지며, 따라서 파일 함수들을 지원하려면 BSD 소켓 역시 VFS inode 자료구조로 표현되어야 한다.

새로 만들어진 BSD socket 자료구조는 주소 패밀리에 따라 특수한 소켓 루틴들에 대한 포 인터를 가지고 있으며, 이는 pops 벡터에서 얻을 수 있는 proto_ops 자료구조에 설정된 다. 타입은 요구한 소켓 타입으로 설정된다. 즉

SOCK_STREAM, SOCK_DGRAM 등등 중의 하 나이다. 주소 패밀리에 따라 다른 생성 함수를 proto_ops 자료구조에 있는 주소를 이용하 여 호출한다.

텅빈 파일 기술자(descriptor)가 현재 프로세스의 fd 벡터에서 할당되고, 이를 가리키는file 자료구조가 초기화된다. 이는 파일 함수 포인터가 BSD 소켓 인터페이스에서 지원하는 BSD 소켓 파일 함수들을 가리키도록 설정하는 것을 포함한다. 이후의 작업들은 소켓 인터페이스 로 전달되고 인터페이스는 차례로 주소 패밀리의 함수들을 호출함으로써 이들을 지원하는 주소 패밀리로 전달한다.

10.4.2 주소와 INET BSD 소켓을 바인드하기(binding)

들어오는 인터넷 접속 요구를 기다릴 수 (listen) 있으려면, 각 서버는 INET BSD 소켓을 만 들어 이를 서버의 주소와 바인드해 주어야 한다. 이 바인드 작업은 대부분 INET 소켓계층 이 아래 계층인 TCP와 UDP 프로토콜 계층으로부터 어느 정도 지원을 받아 처리한다. 주소 와 바인드 되어있는 소켓은 다른 통신을 위해서 사용할 수 없다. 이는 socket의 상태는 TCP_CLOSE여야만 한다는 것을 말한다. 바인드 함수에 전달된 sockaddr은 바인드할 IP 주소와, 옵션으로 포트 번호를 가지고 있다. 보통은 INET 주소 패밀리를 지원하며 위에서 이 인터페이스를 사용할 수 있는 네트워크 장치에 할당된 IP 주소가, 여기서 바인드 되는 IP 주소이다. 현재 시스템에서 어떤 네트워크 인터페이스가 활성화되어 있는지는 ifconfig 명령을 사용하여 알 수 있다. IP 주소는 모두 1이거나 모두 0인 IP 브로드캐스트(broadcast) 주소일 수도 있다. 이들은 특별한 주소로서 "모든사람에게 보내라"를 의미한다. 또, 기계가 투명한 프록시나 방화벽으로 동작하고 있다면, 어떤 IP 주소하고도 바인드할 수 있다. 그러 나 슈퍼유저 권한을 가진 프로세스만이 아무 IP 주소에나 바인드 할 수 있다. 바인드된 IP 주소는 recv_addr에 있는 sock 자료구조와 saddr 항목에 저장된다. 이들은 해시로 찾을 때 쓰이며, 보내는 IP 주소로도 쓰인다. 포트 번호는 옵션이며 이를 지정하지 않으면 이를 지원하는 네트워크에게 아무것이나 비어있는 것을 달라고 요청한다. 관습적으로 1024보다 작은 포트번호는 슈퍼유저 권한을 가지지 않은 프로세스는 사용할 수 없다. 만약 아래의 네 트워크 계층에서 포트 번호를 할당한다면, 이는 항상 1024보다 큰 것을 할당할 것이다.

아래기반의 네트워크 장치는 패킷을 받으면, 이를 올바른 INET과 BSD 소켓으로 전달하여 처리될 수 있도록 해야 한다. 이런 이유로 UDP와 TCP는 들어온 IP 메시지에 있는 주소를 조회하여 올바른 socket/sock 쌍으로 전달하는데 사용할 수 있도록 해시 테이블을 관리한 다. TCP는 연결 지향 프로토콜이므로 UDP 패킷을 처리할 때보다 TCP 패킷을 처리하는데 더 많은 정보가 사용된다.

UDP는 할당된 UDP 포트의 해시 테이블인 udp_hash 테이블을 관리한다. 이는 sock 자료 구조의 포인터로서 포트 번호에 기반한 해시 함수로 인덱스되어 있다. UDP 해시 테이블은 허용되는 포트 번호의 수보다는 훨씬 적으므로 (udp_hash는 128 또는 UDP_HTABLE_SIZE 의 값 만큼의 엔트리를 갖는다), 테이블의 어떤 엔트리들은 sock 자료구조의 연결 고리(이들은 sock의 next 포인터로 서로 연결된다)를 가리킨다.

TCP는 여러 개의 해시 테이블을 관리하므로 훨씬 더 복잡하다. 어쨌든 TCP는 바인드 작업 동안에 바인드하는 sock 자료구조를 이의 해시 테이블에 실제로 추가하지는 않고, 단지 요 구한 포트번호가 현재 사용되고 있는지만 검사한다. sock 자료구조는 리슨 작업을 하는 도 중에 TCP의 해시 테이블에 추가된다

REVIEW NOTE : 입력한 루트는 어떻게 되는가?

10.4.3 INET BSD 소켓으로 연결하기

소켓이 만들어지고, 이것이 내부로의 연결 요구를 받기 위한 용도로 사용되지 않았다면, 이 는 외부로의 연결 요구에 사용할 수 있다. UDP와 같은 비연결지향 프로토콜(connectionless protocol)에서는 이런 작업은 별로 하는 일이 없지만, TCP같은 연결지향 프로토콜(connection oriented protocol)에서는 이는 두 개의 응용프로그램간에 가상 회로를 만드는 것을 포함한다.

외부로의 연결은 적절한 상태에 있는 INET BSD 소켓에서만 이루어질 수 있다 : 말하자면 이미 연결이 되어 있거나, 내부로의 연결을 기다리는데 사용하고 있는 것은 안된다는 것이 다. 이는 BSD 소켓 자료구조가 SS_UNCONNECTED 상태에 있다는 것을 의미한다. UDP 프 로토콜은 응용프로그램간에 가상 연결을 만들지 않는다. 보내는 메시지들은 모두 데이터그 램이며, 메시지의 한 부분이 목적지에 도착할 수도, 도착하지 않을 수도 있다. 그렇긴 하지 만, 접속 BSD 소켓 함수를 지원한다. UDP INET BSD 소켓에서의 접속 작업은 단순히 원격 응용프로그램의 주소 - IP 주소와 포트 번호 - 를 설정할 뿐이다. 추가적으로 라우팅 테이블 엔트리에 대한 캐시를 셋업하여, 이 BSD 소켓으로 보낸 UDP 패킷이 다시 라우팅 데

이터베 이스를 검사할 필요가 없도록 (이 루트가 틀린 것이 되기 전까지는) 한다. 캐시된 라우팅 정 보는 INET sock 자료구조에서 ip_route_cache가 가리키고 있다. 만약 아무런 주소 정보 도 지정하지 않는다면, 이 캐시된 라우팅과 IP 주소 정보를 자동으로 BSD 소켓을 사용하여 보내는 메시지에 사용한다. UDP는 sock의 상태를 TCP_ESTABLISHED로 바꾼다.

TCP BSD 소켓에서의 접속 작업에서는, TCP는 접속 정보를 가진 TCP 메시지를 하나 만들어 서 이를 주어진 IP 목적지로 보내야 한다. 이 TCP 메시지는 접속에 관련된 갖가지 정보들을 가지고 있다. 유일한 시작 메시지 순서 번호와 시작하는 (initiator) 호스트에서 처리할 수 있 는 메시지의 최대 크기, 보내고 받는 윈도우 크기, 등등이 그것이다. TCP에서는 모든 메시 지에 번호가 붙으며, 초기 순서 번호는 첫번째 메시지 번호에 사용한다. 리눅스는 악의적인 프로토콜 공격을 피하기 위해 허용하는 범위 내에서 임의의 값을 고른다. 한쪽에서 전송한 메시지를 다른 쪽에서 성공적으로 받으면, 모든 메시지에 대해 그것이 성공적으로 깨지지 않고 도착했다는 것을 말하는 응답해 주어야 한다. 응답받지 않은 메시지는 다시 보내게 된 다. 송수신 윈도우 크기는 응답을 보내지 않고 있을 수 있는 메시지의 수이다 (이만큼의 메 시지를 보낼 때까지 ACK가 오지 않아도 된다). 최대 메시지 크기는 요청을 시작한 쪽에서 사용하고 있는 네트워크 장치에 따른다. 만약 받는 쪽의 네트워크 장치가 이보다 작은 최대 메시지 크기를 지원한다면, 접속에서는 둘 중에 최소값을 사용하게 된다. 밖으로의 TCP 접 속 요청을 하는 응용프로그램은 대상 응용프로그램이 이 접속 요구를 받거나 거부한다는 응 답을 보낼 때까지 기다려야 한다. TCP sock은 이제 메시지가 들어오길 기다려야 하므로, tcp_listening_hash를 추가하여, 들어오는 TCP 메시지가 sock 자료구조로 갈 수 있게 한다. TCP는 또한 대상 응용프로그램이 요구에 응답을 보내주지 않는 경우 밖으로의 접속 요구를 타임아웃 할 수 있도록 타이머를 시작한다.

10.4.4 INET BSD 소켓에서 리슨(listening)

소켓에 주소를 바인드 하였다면, 바인드한 주소를 지정하여 들어오는 접속 요구를 기다릴 수 있다. 네트워크 응용프로그램은 먼저 주소를 바인드 하지 않고도 접속을 기다릴 수 있는 데, 이런 경우 INET 소켓 계층은 지금 프로토콜에서 사용하지 않고 있는 포트 번호를 찾아 이를 소켓에 자동으로 바인드 해준다. 리슨 소켓 함수는 소켓의 상태를 TCP_LISTEN으로 바꾸고 들어오는 접속을 허가하는데 필요한 네트워크 특수 작업들을 한다.

UDP 소켓에 있어서는 소켓의 상태를 바꾸는 것으로도 충분하지만, TCP는 소켓의 sock 자 료구조를 두개의 해시 테이블에 추가하여 활성화되도록 한다. 이 두 개의 해시 테이블은 tcp_bound_hash와 tcp_listening_hash 테이블이다. 둘 다 IP 포트 번호에 기반한 해 시 함수를 통하여 인덱스되어 있다.

활성화된 리슨 소켓에 대해 TCP 접속 요구가 들어오면, TCP는 이를 나타내기 위해 새로운 sock 자료구조를 만든다. 이 sock 자료구조는 이 TCP 접속이 결국 받아들여진다면 TCP 접속의 하반부가 된다. 또한 접속 요구를 포함하고 있는 들어온 sk_buff를 복사하여, 기다 리는 sock 자료구조의 receive_queue의 뒤에 이를 추가한다. 복사한 sk_buff는 새로 만든 sock 자료구조에 대한 포인터를 갖는다.

10.4.5 접속 요구 허가하기(accepting)

UDP는 접속이라는 개념을 지원하지 않으므로, INET 소켓 접속을 허락하는 것은 TCP 프로 토콜에만 적용이 되며, 접속을 기다리는 소켓에서 접속을 허락하는 것은 원래의 기다리는 소켓에서 socket 자료구조를 복사하여 새로운 socket을 만든다. 허가 작업은 자신을 지 원하는 프로토콜 계층, 이 경우 INET 계층으로 넘어가서 들어오는 어떤 접속 요구를 받아 들이라고 한다. 만약 아래 계층의 프로토콜이 UDP같이 접속을 지원하지 않는 것이라면 이 접속 허가 과정은 실패한다. 그렇지 않으면 접속 허가 과정은 실제 프로토콜, 이 경우 TCP 로 전달된다. 이 접속 허가 작업은 블럭킹 모드일수도, 블럭킹 모드가 아닐수도 있다. 블럭 킹 모드가 아닌 경우, 만약 아무런 들어오는 접속이 없으면, 이 접속 작업은 실패하고, 새로 만들어진 socket 자료구조는 버려질 것이다. 블럭킹 모드인 경우, 접속 허가를 하는 네트 워크 응용프로그램은 대기 큐에 들어가고 TCP 접속 요구를 받을 때까지 중단된다. 접속 요 구가 들어오면, 그 요구를 갖고 있는 sk_buff는 무시되고, sock 자료구조는 이전에 만든 새 socket 자료구조와 연결되어 있는 INET 소켓 계층으로 되돌아간다. 네트워크 응용프로 그램에 새로 만들어진 소켓의 파일 기술자(fd)를 돌려주고, 응용 프로그램은 새로 만들어진 BSD 소켓을 가지고 소켓 작업을 하는데 이 파일 기술자를 사용할 수 있다.

10.5 IP 계층

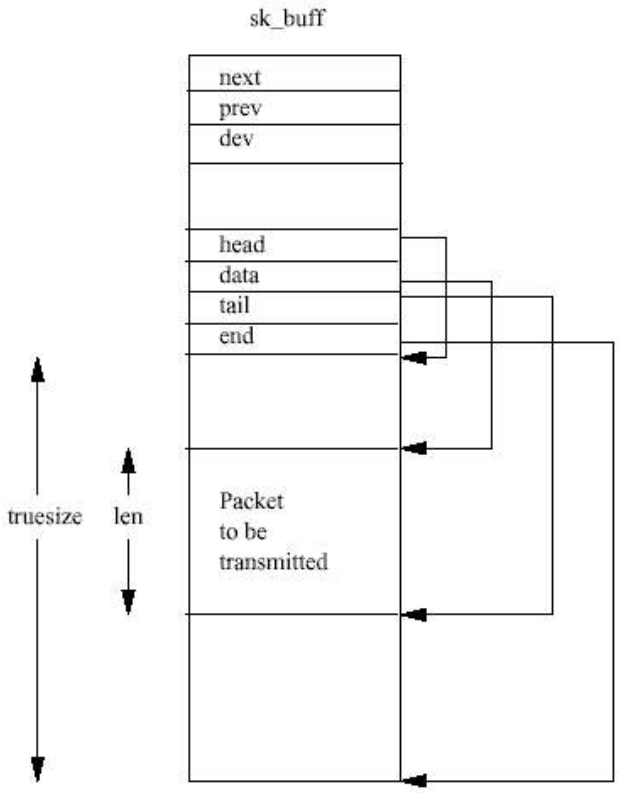

그림 10.4 : 소켓 버퍼 (sk_buff)

10.5.1 소켓 버퍼(Socket Buffer)

많은 네트워크 프로토콜 계층을 가지고, 각각이 다른 것의 서비스를 사용하는 방법의 문제 중의 하나는, 각 프로토콜이 전송하는 데이터에 프로토콜 헤더와 꼬리를 붙이고, 받은 데이 터를 처리할 때 이를 제거해야 한다는 것이다. 이는 각 프로토콜 계층마다 특별한 프로토콜 헤더와 꼬리를 찾아야 하므로 프로토콜 사이에 데이터 버퍼를 전달하는 것을 어렵게 만든다. 방법중의 하나는 각 계층마다 버퍼를 복사하는 것이지만, 이는 매우 비효율적이다. 대신 리 눅스는 프로토콜 계층 사이와 네트워크 디바이스 드라이버 간에 데이터를 전달하기 위해 sk_buffs라는 소켓 버퍼를 사용한다. sk_buffs는 포인터와 길이 항목을 가지고 있어서 각 프로토콜 계층이 표준 함수를 통해 응용프로그램 데이터를 다룰 수 있게 한다.

그림 10.4는 sk_buff 자료구조를 보여준다. 각 sk_buff는 자신과 연관된 데이터 블럭을 가지고 있다. sk_buff는 네개의 데이터 포인터를 가지고 있는데, 이들은 소켓 버퍼 데이터 를 다루고 관리하는데 사용된다.

- **헤드(head)** 메모리에서 데이터의 시작을 가리킨다. 이는 sk_buff와 이와 관련된 데이터 블 럭을 할당할 때 고정된다.
- **데이터(data)** 현재 프로토콜 데이터의 시작을 가리킨다. 이 포인터는 현재 sk_buff를 소 유하고 있는 프로토콜 계층에 따라 달라진다.
- **꼬리(tail)** 현재 프로토콜 데이터의 끝을 가리킨다. 마찬가지로, 소유하고 있는 프로토콜 계 층에 따라 달라진다.
- **끝(end)** 메모리에서 데이터 영역의 끝을 가리킨다. sk_buff를 할당할 때 결정된다.

길이를 나타내는 항목으로는 len과 truesize 두개가 있으며, 이들은 각각 현재 프로토콜 패킷의 길이와, 상대적인 데이터 버퍼의 전체 크기를 나타낸다. sk_buff를 다루는 코드는 응용프로그램 데이터에 프로토콜 헤더와 꼬리를 붙이고 제거하는 표준적인 방법들을 제공한 다. 이들은 안전하게 sk_buff에 있는 data, tail, 그리고 len 항목들을 다룬다.

- **push** data 포인터를 데이터 영역의 시작쪽으로 이동하고, len 항목을 증가시킨다. 이는 전 송할 데이터의 시작부분에 데이터나 프로토콜 헤더를 붙이는데 사용된다.

- **pull** data 포인터를 시작부분에서 먼 쪽으로, 데이터 영역의 끝쪽으로 이동하고, len 항목을 감소시킨다. 이는 수신한 데이터의 시작부분에서 데이터나 프로토콜 헤더를 제거하는데 사용된다.
- **put** tail 포인터를 데이터 영역의 끝쪽으로 이동하고 len 항목을 증가시킨다. 이는 전송할 데이터의 끝에 데이터나 프로토콜 정보를 추가하는데 사용된다.
- **trim** tail 포인터를 데이터 영역의 시작쪽으로 이동하고 len 항목을 감소시킨다. 이는 수 신한 패킷에서 데이터나 프로토콜 꼬리를 제거하는데 사용된다.

sk_buff 자료구조는 또한 처리도중에 sk_buff 의 이중 원형 연결 리스트에 저장하는데 사용하는 포인터들을 가지고 있다. 그리고 sk_buffs를 이들 리스트의 앞이나 뒤에 추가하 고 제거하는데 사용하는 일반적인 sk_buff 루틴들도 있다.

10.5.2 IP 패킷 수신하기

커널에서 리눅스 드라이버들이 어떻게 만들어지고 초기화되는지는 8장에서 설명했다. 이 초 기화의 결과는 dev_base 리스트에서 서로 연결되어 있는 일련의 device 자료구조이다. 각 device 자료구조는 장치를 서술하고, 네트워크 프로토콜 계층에서 네트워크 드라이버가 어떤 일을 수행해야 할 때 부를 수 있는 콜백 루틴 세트를 제공한다. 이들 함수들은 대부분 데이터 전송과 네트워크 장치의 주소에 관련되어 있다. 네트워크 장치가 네트워크로부터 패 킷을 수신하면 이 수신한 데이터를 sk_buff 자료구조로 바꾸어야 한다. 네트워크 드라이버 는 이들을 수신할 때마다 backlog 큐에 수신한 sk_buff 들을 추가한다. 만약 backlog 큐가 너무 커지면, 수신한 sk_buff 들은 무시된다. 이제 해야 할 일이 있으므로 실행할 준 비가 되었다고 네트워크 하반부(bottom half)에 표시한다.

스케줄러가 네트워크 하반부 핸들러를 실행하면, 이는sk_buff의 backlog 큐를 처리하기 이전에 수신한 패킷을 어떤 프로토콜 계층으로 전달할지를 결정하며 전송되길 기다리고 있 는 네트워크 패킷들을 처리한다. 리눅스 네트워킹 계층을 초기화할 때 각 프로토콜은 packet_type 자료구조를 ptype_all 리스트나 ptype_base 해시테이블에 추가함으로서 자신들을 등록했다. packet_type 자료구조는 프로토콜 타입과 네트워크 장치에 대한 포인 터, 프로토콜의 수신 데이터 처리 루틴, 그리고 마지막으로 리스트나 해시 고리에 있는 다음 packet_type 자료구조에 대한 포인터를 가지고 있다. ptype_all 고리는 어떤 네트워크 장치이든지부터 수신되는 모든 패킷들을 엿보는데(snoop) 사용되지만 잘 사용되지 않는다. ptype_base 해시 테이블은 프로토콜 식별자로 해시되어 있으며, 들어오는 네트워크 패킷 을 어떤 프로토콜이 받을 것인지 결정하는데 사용된다. 네트워크 하반부는 들어오는 sk_buff의 프로토콜 타입과 각 테이블에 있는 하나 이상의 packet_type 엔트리와 매치 시킨다. 프로토콜은 하나 이상의 엔트리와 매치될 수 있는데, 예를 들어 모든 네트워크 트래픽을 엿볼 때 같은 경우이며, 이 경우 sk_buff는 복제가 된다. sk_buff는 매치되는 프로 토콜 처리 루틴으로 전달된다.

10.5.3 IP 패킷 전송하기

패킷은 응용프로그램이 데이터를 교환하거나, 네트워크 프로토콜이 이미 만들어진 연결이나 만들어지는 연결을 지원할 때 만들어져서 보내진다. 어떤 방법으로 데이터가 만들어졌던지 간에 데이터를 포함하고 있는 sk_buff가 만들어지고, 각 프로토콜 계층을 통과하면서 프 로토콜 계층이 다양한 헤더를 붙인다.

sk_buff는 전송할 네트워크 장치로 전달되어야 한다. 먼저 IP 같은 프로토콜이라도 어떤 네트워크 장치를 사용할지를 결정해야 한다. 이는 패킷에 가장 맞는 루트에 따라 다르다. PPP 프로토콜같은 것을 통해 모뎀으로 하나의 네트워크에 연결된 컴퓨터에 있어서는 이 루 트를 선택하는 것은 쉽다. 패킷은 루프백 장치를 통해 로컬호스트나, PPP 모뎀 연결의 끝에 있는 게이트웨이 둘 중 하나로 전송될 것이다. 이더넷으로 연결되어 있는 컴퓨터에 있어서 는, 네트워크에 많은 컴퓨터가 연결되어 있으므로 이 선택은 더 어렵다.

IP 패킷을 전송할 때 항상 IP는 도달할 IP 주소로 가는 루트(route)를 해결하기 위해 라우팅 테이블(routing table)을 사용한다. 각 IP 목적지는 라우팅 테이블에서 성공적으로 찾게 되어, 사용할 루트를 기술하는 rtable 자료구조를 돌려준다. 이는 사용할 출발지 IP 주소와, 네트 워크 device 자료구조의 주소, 때때로 미리 만들어진 하드웨어 헤더를 포함한다. 이 하드 웨어 헤더는 네트워크 장치마다 다른 것으로서 출발지와 도착지의 하드웨어 주소와, 매개체 별로 다른 정보를 가지고 있다. 만약 네트워크 장치가 이더넷 장치이라면, 하드웨어 헤더는 그림 10.1에서 보는 바와 같을 것이며, 출발지와 도착지 주소는 물리적인 이더넷 주소일 것 이다. 하드웨어 헤더는 루트와 함께 캐시되는데, 이는 이 하드웨어 헤더가 이 루트를 통하여 전송하는 모든 IP 패킷에 추가되어야 하는데, 이를 다시 만드는 것은 시간이 걸리기 때문이 다. 하드웨어 헤더는 ARP 프로토콜로 해결되어야 하는 물리적인 주소를 가질 수도 있다. 이 경우 밖으로 나가는 패킷은 주소가 해결될 때까지

꼼짝못하고 기다리고 있어야 한다. 한번 주소가 해결되고 나면, 하드웨어 헤더가 만들어지고, 이 인터페이스를 사용하는 IP 패킷이 다시 ARP를 할 필요가 없도록 이 하드웨어 헤더를 캐시한다.

10.5.4 데이터 조각내기 (data fragmentation)

모든 네트워크 장치는 최대 패킷 크기를 가지고 있으며, 이보다 큰 크기의 데이터를 보내거 나 받을 수 없다. IP 프로토콜은 이런 경우를 허용하여 데이터를 네트워크 장치가 처리할 수 있는 패킷 크기로 데이터를 잘게 쪼갠다. IP 프로토콜 헤더는 플래그와 이 조각의 옵셋을 담 은 조각 항목을 가지고 있다.

IP 패킷이 전송할 준비가 되면, IP는 IP 패킷을 밖으로 보낼 네트워크 장치를 찾는다. 장치 는 IP 라우팅 테이블에서 찾게 된다. 각 device는 최대 전송 단위를 나타내는 항목으로 가 지고 있는데 (바이트 단위), 이는 mtu 항목이다. 만약 장치의 mtu가 전송하려는 IP 패킷의 크기보다 작으면, IP 패킷은 좀 더 작은 크기(mtu 크기)의 조각으로 쪼개져야 한다. 각 조각은 sk_buff로 표현된다. IP 헤더에는 이것이 조각이며, 이 패킷이 데이터의 어떤 옵셋부터 가지고 있는지 표시된다. 마지막 패킷은 마지막 IP 조각이라고 표시된다. 만약, 이 쪼개는 도중에 IP가 sk_buff를 할당받지 못한다면 전송을 실패하게 된다.

IP 조각을 수신하는 것은 전송하는 것보다 더 어려운데, 이는 IP 조각이 아무런 순서로나 도착할 수 있으므로 모두 수신 받아야 재조립할 수 있기 때문이다. IP 패킷을 수신할 때마다 이것이 IP 조각인지 검사한다. 메시지 조각이 처음 도착하면, IP는 새 ipq 자료구조를 만들 고, 이를 재조립을 기다리는 IP 조각의 리스트인 ipqueue에 연결한다. IP 조각이 계속 수신 되면 맞는 ipq 자료구조를 찾아 이 조각을 나타낼 ipfrag 자료구조를 새로 만든다. 각 ipq 자료구조는 조각난 IP 수신 프레임을 출발지와 도착지 IP 주소와 함께 유일하게 기술 하며, 위 계층 프로토콜 식별자와 이 IP 프레임의 식별자를 기술한다. 모든 조각이 도착하면, 이들은 하나의 sk_buff로 합쳐지고 처리할 다음 프로토콜 계층으로 전달된다. 각 ipq는 제대로 된 조각이 도착할 때마다 다시 시작되는 타이머를 가지고 있다. 만약 이 타이머가 만료되면, ipq 자료구조와 이것의 ipfrag들은 소멸되며, 메시지는 전송 중에 사라진 것으 로 간주된다. 이 메시지를 다시 전송하는 것은 더 윗 레벨의 프로토콜이 담당하는 문제이다.

10.6 주소 결정 프로토콜(Address Resolution Protocol, ARP)

주소 결정 프로토콜의 역할은 IP 주소에서 이더넷 주소와 같은 물리적 하드웨어 주소로의 변환을 제공하는 것이다. IP는 데이터를 전송할 디바이스 드라이버에게 전달하기 (sk_buff 의 형태로) 바로 전에 이런 변환을 필요로 한다. 이는 이 장치가 하드웨어 헤더를 필요로 하는지, 만약 그렇다면 이 패킷용으로 하드웨어 헤더를 다시 만들어야 하는지 알기 위해 여러가지 검사를 수행한다. 리눅스는 하드웨어 헤더를 자주 다시 만들지 않도록 이를 캐시한 다. 만약 하드웨어 헤더를 다시 만들 필요가 있다면, 장치 고유의 하드웨어 헤더 재제작 루 틴을 호출한다. 모든 이더넷 장치는 똑같은 일반적인 헤더 재제작 루틴을 사용하며, 이 루틴 은 목적지 IP 주소를 물리적인 주소로 바꾸기 위해 차례로 ARP 서비스를 사용한다.

ARP 프로토콜 그 자체는 매우 단순하며, ARP 요구와 ARP 응답 두가지 메시지 형태로 이루 어져 있다. ARP 요구는 변환을 필요로 하는 IP 주소를 가지고 있고, 응답은 (바라건데) 변환 된 IP 주소인 하드웨어 주소를 가지고 있다. ARP 요구는 네트워크에 연결된 모든 호스트로 방송(브로드캐스트) 되므로, 이더넷 네트워크에서는 이더넷에 연결된 모든 기계들이 이 ARP 요구를 받게 된다. 이 요구에 있는 IP 주소를 소유하고 있는 기계는 이 ARP 요구에 응답하 여 자신의 물리적인 주소를 담고 있는 ARP 응답으로 답하게 된다.

리눅스에서 ARP 프로토콜 계층은 각각 IP에서 물리주소로의 변환을 나타내는 arp_table 자료구조의 테이블을 가지고 이루어져 있다. 이들 엔트리들은 IP주소가 변환될 필요가 있을 때 만들어지고, 시간이 지나 낡아지면 제거된다. 각 arp_table 자료구조는 다음과 같은 항 목들을 가진다 :

```
마지막 사용(last used)      ARP 엔트리가 마지막으로 사용된 시간
마지막 갱신(last updated)   ARP 엔트리가 마지막으로 갱신된 시간
플래그(flags)               엔트리가 완료되었는지 같은 엔트리의 상태를 나타낸다.
IP 주소                     엔트리가 나타내는 IP 주소
하드웨어 주소               변환된 하드웨어 주소
하드웨어 헤더               캐시된 하드웨어 헤더에 대한 포인터
타이머(timer)               응답하지 않는ARP 요구를 타임아웃 시키는데 사용하는
                            timer_list 엔트리
재시도(retries)             이 ARP 요구를 재시도한 횟수
sk_buff 큐                  이 IP 주소를 해결하기 기다리는 sk_buff 엔트리의 리스트
```

ARP 테이블은 arp_table 엔트리들을 잇기 위해 포인터의 테이블로 되어 있다 (arp_tables 벡터). 엔트리들은 이들에 대한 접근 속도를 높이기 위해 캐시되며, 각 엔트 리는 IP 주소의 끝 두 바이트를 가져와 테이블에 대한 인덱스를 계산하고, 원하는 것을 찾 을 때까지 해시 테이블에서 엔트리의 고리를 따라가 찾게 된다. 리눅스는 또한 미리 만들어 진 하드웨어 헤더를 hh_cache 자료구조 형태로 arp_table 엔트리에 캐시시킨다.

IP 주소변환을 요구했는데 일치하는 arp_table 엔트리가 없을 경우, ARP는 ARP 요구 메 시지를 보내야 한다. ARP는 arp_table에서 새 arp_table 엔트리를 만들고, 주소 변환을 필요로 하는 패킷들을 포함하고 있는 sk_buff를 새로 만들어진 엔트리의 sk_buff 큐에 큐시킨다. ARP는 ARP 요구를 보내고 ARP 만료 타이머를 실행한다. 아무런 응답이 없다면 ARP는 여러번 재시도를 하고, 여전히 응답이 없다면 ARP는 arp_table 엔트리를 제거한 다. IP 주소가 변환되기를 기다려 큐되어 있는 어떤 sk_buff 자료구조이든 간에 통지를 받 게 되고, 이런 실패와 협조하는 것은 이들을 전송하려는 프로토콜 계층의 몫이다. UDP는 잃 어버린 패킷에 대해서 신경을 쓰지 않지만, TCP는 성립된 TCP 링크를 통하여 재전송하려고 시도할 것이다. 만약 IP 주소의 소유자가 하드웨어 주소를 돌려주며 응답한다면, arp_table 엔트리는 완료된 것으로 표시되고, 큐되어 있는 모든 sk_buff들은 큐에서 재 거되고 전송될 것이다. 하드웨어 주소는 각 sk_buff의 하드웨어 헤더에 기록된다.

ARP 프로토콜 계층은 자신의 IP 주소를 지정하고 있는 ARP 요구에 반드시 응답해야 한다. 이 계층은 자신의 프로토콜 타입 (ETH_P_ARP)를 등록하고, packet_type 자료구조를 생성 한다. 이는 네트워크 장치가 수신한 모든 ARP 패킷을 전달받게 된다는 것을 의미한다. 이는 ARP 응답뿐만 아니라 ARP 요구도 포함한다. 이는 수신한 장치의 device 자료구조에 저장되어 있는 하드웨어 주소를 사용하여 ARP 응답을 만든다.

네트워크 구성은 시간이 지나면서 변할 수 있으며, IP 주소는 다른 하드웨어 주소로 다시 할 당될 수도 있다. 예를 들어, 어떤 전화접속 서비스는 연결이 될 때마다 각각 다른 IP 주소를 배정한다. ARP 테이블이 가장 최근의 엔트리를 가질 수 있도록, ARP는 정기적인 타이머를 돌려서 모든 arp_table 엔트리들이 타임아웃이 되지 않았는지 살펴본다. 이는 하나 이상 의 캐시된 하드웨어 헤더를 갖고 있는 엔트리들을 제거하지 않도록 매우 조심한다. 이들 엔 트리를 지우는 것은 다른 자료구조들이 이에 의존하고 있으므로 매우 위험하다. 어떤 arp_table 엔트리들은 영구적이며, 이들은 할당이 해제되지 않도록 표시가 된다. ARP 테 이블은 너무 커지면 안된다. 각 arp_table 엔트리는 어느정도 커널 메모리를 잡아먹기 때 문이다. 새 엔트리가 할당되어야 하고 ARP 테이블이 최대 크기에 도달할 때마다, 테이블은 가장 오래된 엔트리들을 찾아 이를 제거한다.

10.7 IP 라우팅(routing)

IP 라우팅 함수는 특정 IP 주소를 목적지로 가진 IP 패킷을 어디로 보낼지를 결정한다. IP 패킷을 전송할 때 많은 선택을 할 수 있다. 목적지에 결국 도착할 수 있을까? 만약 도착할 수 있다면, 전송하는데 어떤 네트워크 장치를 사용할 것인가? 목적지에 도착하는데 사용할 수 있는 네트워크 장치가 하나 이상 있다면, 어떤 것이 더 좋은 것인가? IP 라우팅 데이터베이스는 이들 질문에 대답할 수 있는 정보를 관리한다. 여기에 두가지 데이터베이스가 있는 데, 가장 중요한 것은 전달 정보 데이터베이스(Forwarding Information Database)이다. 이것은 IP 주소와 가장 좋은 길에 대해서 알려진 것들의 소모적인 목록이다. IP 목적지로의 길을 빨 리 찾기 위해, 더 작고 더 빠른 데이터베이스인 루트 캐시(route cache)가 사용된다. 다른 모 든 캐시처럼 이는 자주 접근하는 길들에 대해서만 가지고 있어야 한다; 이것의 내용은 전달 정보 데이터베이스에서 가져온 것이다.

루트는 BSD 소켓 인터페이스로 IOCTL 요구를 보냄으로써 추가되거나 삭제된다. 이들은 프 로토콜에서 프로세스로 전달된다. INET 프로토콜 계층은 IP 루트를 추가하거나 삭제하는데 슈퍼유저 권환을 가진 프로세스만을 허가한다. 이들 루트들은 고정될 수도 있고, 시간이 지 나면서 동적으로 변할 수도 있다. 대부분의 시스템은 라우터가 아니라면 고정된 루트를 사 용한다. 라우터는 지속적으로 모든 알려진 IP 목적지로 가는 길들의 유효성을 검사하는 라 우팅 프로토콜을 실행한다. 라우터가 아닌 시스템들은 단말 시스템이라고 한다. 라우팅 프로 토콜은 GATED같은 데몬으로 구현되어 있으며, 마찬가지로 IOCTL BSD 소켓 인터페이스를 통하여 루트를 추가하거나 삭제한다.

10.7.1 루트 캐시(Route Cache)

IP 루트를 조회하면 일치하는 루트를 찾기 위해 루트 캐시를 먼저 검사한다. 루트 캐시에 일치하는 루트가 없다면 전달 정보 데이터베이스에서 루트를 찾는다. 만약 아무런 루트도 찾을 수 없다면, IP 패킷은 전송에 실패하고 이를 응용프로그램에 알린다. 만약 루트가 전달 정보 데이터베이스에 있고 루트 캐시에 없다면, 이 루트에 해당하는 새 엔트리를 만들어 루 트 캐시에 추가한다. 루트 캐시는 rtable 자료구조의 연결고리에 대한 포인터를 가지고 있 는 테이블(ip_rt_hash_table)

이다. 루트 테이블에서의 인덱스는 IP 주소의 하단 두 바이 트에 기반한 해시함수이다. 이들 두 바이트는 목적지마다 가장 달라서 해시값을 가장 잘 분 산시켜 줄 수 있는 것이다. 각 rtable 엔트리는 루트에 대한 정보 - 목적 IP 주소와 이 IP 주소에 도달하는데 사용할 네트워크 device, 사용할 수 있는 메시지의 최대 크기 등등 - 를 가지고 있다. 이는 또한 참조 횟수도 가지고 있는데, 이는 사용횟수와 이것이 사용된 마지 막 시간의 타임스탬프를 가지고 있다 (jiffies 값으로). 참조횟수는 이 루트가 사용될 때 마다 증가하여, 이 루트를 사용하는 네트워크 연결의 숫자를 보여준다. 이는 응용프로그램이 이 루트를 사용하기를 그만두면 감소한다. 사용횟수는 이 루트를 찾았을 때마다 증가하며, rtable 해시 고리에서 이 엔트리의 순서를 결정하는데 사용된다. 루트 캐시에 있는 모든 엔트리에 있는 마지막 사용한 타임스탬프를 정기적으로 검사하여 rtable이 너무 오래되지 않았는지 살핀다. 만약 루트가 최근에 사용되지 않았다면 루트 캐시에서 빠지게 된다. 만약 루트가 루트 캐시에 있다면, 이 루트는 가장 많이 사용한 엔트리가 해시 고리의 맨 앞에 오 도록 배치된다. 이는 루트를 조회할 때 빨리 찾게 된다는 것을 의미한다.

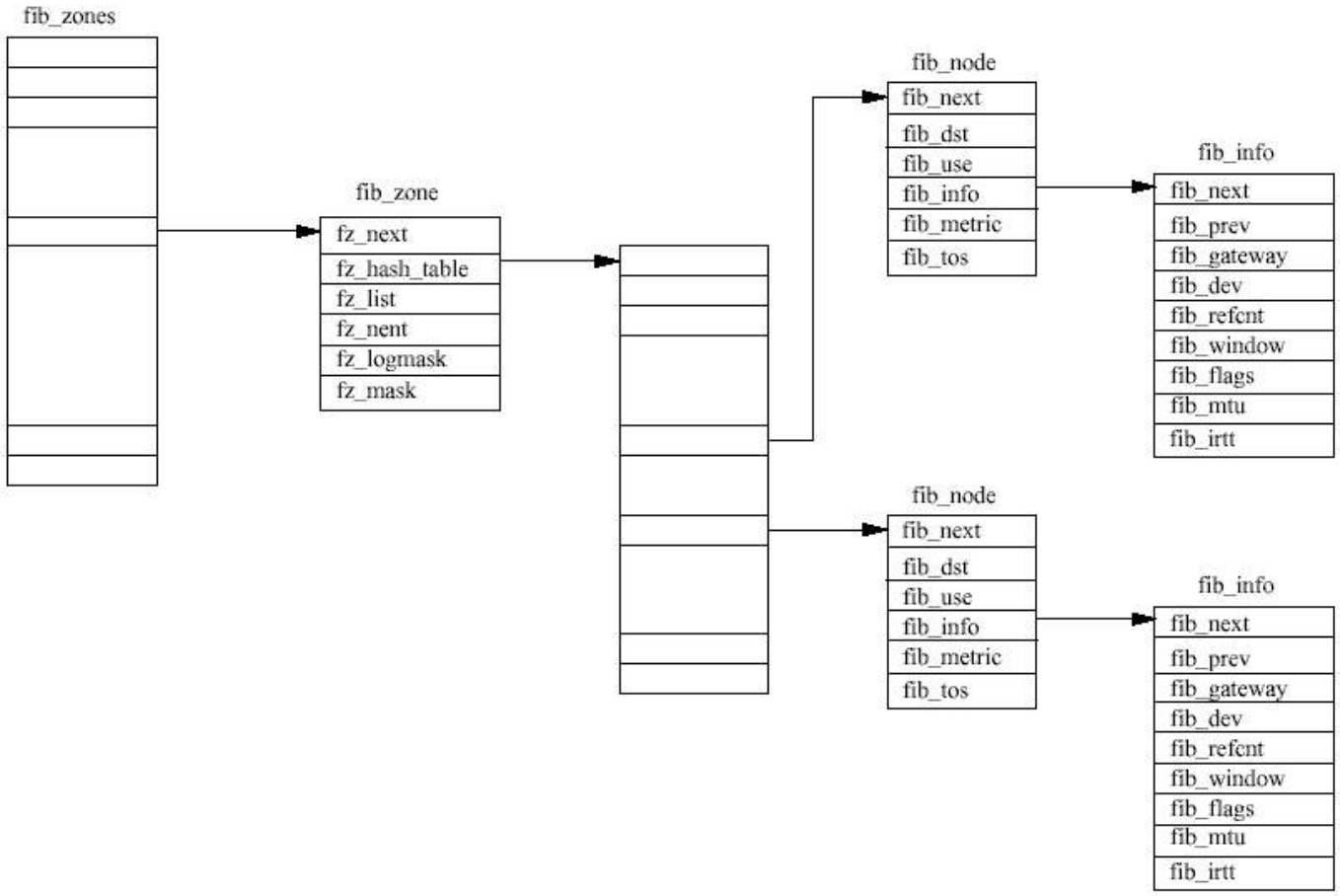

그림 10.5 : 전달 정보 데이터베이스

10.7.2 전달 정보 데이터베이스(Forwarding Information Database)

전달 정보 데이터베이스(그림 10.5에서 보여주고 있다)는 어떤 시간에 시스템에서 사용할 수 있는 루트들을 IP 관점에서 가지고 있다. 이는 매우 복잡한 자료구조이며, 상당히 효과적 으로 배치되어 있지만, 참고하기에 빠른 데이터베이스는 아니다. 특히, 전송하는 모든 IP 패 킷마다 목적지를 이 데이터베이스에서 찾게 된다면 매우 느릴 것이다. 이는 루트 캐시가 있 는 이유이기도 하다 - 알고 있는 좋은 루트를 사용하여 IP 패킷 전송하는 것을 더 빠르게 하기. 루트 캐시는 전달 정보 데이터베이스에서 파생된 것으로 자주 사용하는 엔트리들을 대표한다.

각 IP 서브넷은 fib_zone 자료구조로 표현한다. 이들 모두는 fib_zones 해시 테이블에서 가리키고 있다. 해시 인덱스는 IP 서브넷 매스크에서 만들어진다. 똑같은 서브넷으로의 모든 루트들은 fib_node의 쌍으로 나타내지며, fib_info 자료구조는 각 fib_zone 자료구조 의 fz_list로 큐된다. 만약 이 서브넷에 있는 루트의 숫자가 커지면, fib_node 자료구조 를 쉽게 찾기 위해 해시테이블이 만들어진다.

똑같은 IP 서브넷에 여러개의 루트가 있을 수 있으며, 이들 루트들은 여러 게이트웨이 중의 하나를 통하게 된다. IP 라우팅 계층은 똑같은 게이트웨이를 사용하여 하나의 서브넷으로 여 러 개의 루트가 있는 것을 허가하지 않는다. 다르게 말하면, 서브넷으로 가는 루트가 여러 개가 있다면, 각 루트는 다른 게이트웨이를 사용하도록 하여야 한다는 것이다. 각 루트와 연 관되어 있는 것은 그것의 거리(metric)이다. 이것은 이 경로가 얼마나 유리한지를 측정하게 하는 것이다. 한 루트의 거리는 본질적으로 목적하는 서브넷에 도착하기까지 거쳐야 하는 IP 서브넷의 수이다. 이 값이 더 클 수록 더 좋지 않은 루트이다.

번역 : 김성룡, 이호, 홍경선
정리 : 심마로, 이호

역주 1) 가장 널리 사용되는 웹 서버인 아파치의 절반 이상이 리눅스에서 동작중이다. (심마 로)

2) 국립 과학 재단(National Science Foundation)

역주 3) 이더넷은 방송 프로토콜을 사용하고, 이 때문에 보안성이 떨어지는 측면이 있다 (심 마로)

4) 동기적 읽기 전용 메모리(Synchronous Read Only Memory)