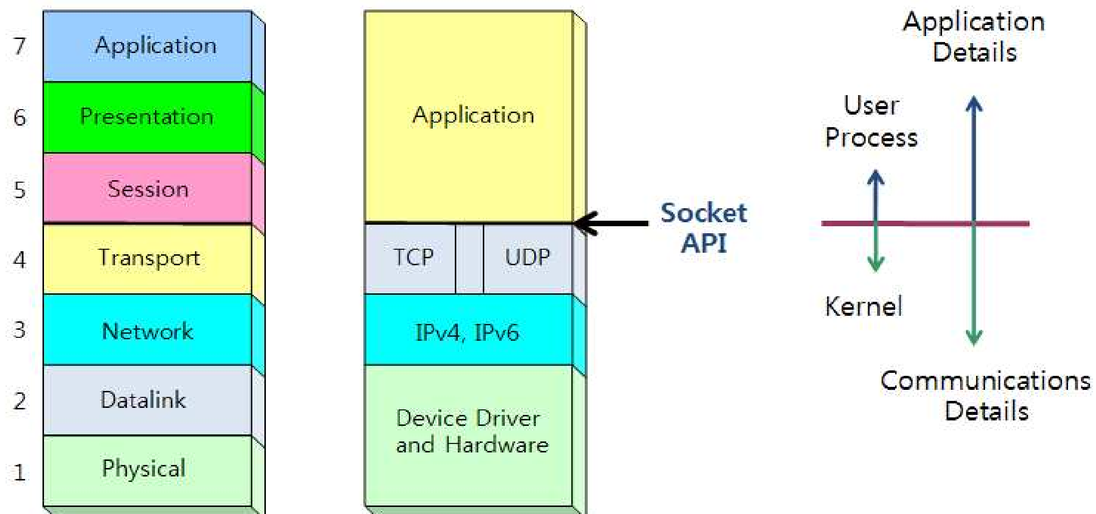
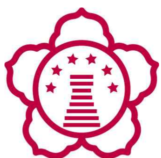
리눅스 네트웍 프로그래밍

(Linux Network Programming)

高碩住 著





경북대학교 전자전기컴퓨터학부

서 문

이 교재는 리눅스 환경에서 네트웍 소켓 프로그래밍 학습을 위한 참조교재로 개발되었다. 최근 유무선 통신망에서 다양한 종류의 인터넷 서비스가 널리 보급되고 있으며, 이에 따라 컴퓨터통신 관련 전공자에게 네트웍 프로그래밍은 필수처럼 여겨지고 있다. 이 교재는 학부과정에서 '데이터통신', '컴퓨터 네트워크' 관련 과목의 보조 교육자료로서 활용될 수 있으며, 특강 형태의 강좌에도 활용될 수 있을 것이다.

이 교재는 "Unix Network Programming: The Sockets Networking API" (R. Stevens 저) 도서의 내용을 토대로 리눅스 플랫폼에 맞는 소켓 API 함수 사용 예제를 기술한다. 내용 자체는 기존 교재와 동일하나 리눅스 환경에 맞도록 수정하였다. 이 교재는 학부과정에서 '컴퓨터 네트워크' 관련 수업을 이수하였거나 이수하고 있는 학생을 대상으로 쓰였다. TCP/IP 인터넷 프로토콜을 처음 접하는 학생들을 위해 부록 A에 인터넷 프로토콜 개요 부분을 정리하였다.

이 교재의 예제 프로그램 코드를 실제 리눅스 플랫폼에서 시험해 볼 수 있으며, 각 예제 코드에서 참조하고 있는 'lnp.h' 헤더 파일은 부록 B와 아래 사이트에서 얻을 수 있다.

http://protocol.knu.ac.kr/pub/lnp.h

이 교재를 통해 네트웍 소켓 프로그래밍에 대한 이해에 조금이나마 도움이 되었으면 하는 바램을 가지며, 끝으로 이 책이 나오기까지 많은 도움을 준 경북대학교 통신프로토콜 연구실의 김상태, 박재성, 이동화, 권순홍, 김지인 군에게 감사의 마음을 전한다.

2009년 3월

경북대학교 전자전기컴퓨터학부

고 석 주

목 차

1. 서론 1

1.1 DAYTIME CLIENT 예제 2

1.2 오류처리 함수와 WRAPPING FUNCTIONS 4

1.3 DAYTIME SERVER 예제 5

1.4 TCP/IP 프로토콜과 소켓 프로그래밍 7

2. 수송계층 프로토콜: TCP, UDP, SCTP 9

2.1 USER DATAGRAM PROTOCOL (UDP) 11

2.2 TRANSMISSION CONTROL PROTOCOL (TCP) 11

2.3 STREAM CONTROL TRANSMISSION PROTOCOL (SCTP) 13

2.4 TCP 연결 설정 및 종료 13

2.5 TIME_WAIT 상태 18

2.6 SCTP ASSOCIATION 설정 및 종료 20

2.7 포트 번호 24

2.8 CONCURRENT 서버 25

2.9 소켓의 송신버퍼 28

3. 기본적인 SOCKETS 33

3.1 SOCKET ADDRESS 구조체 33

3.2 VALUE-RESULT ARGUMENTS 41

3.3 BYTE ORDERING 함수 43

3.4 BYTE MANIPULATION 함수 47

3.5 INET_ATON, INET_ADDR, INET_NTOA 함수 48

3.6 INET_PTON, INET_NTOP 함수 50

4. TCP SOCKETS 53

4.1 SOCKET 함수 54

4.2 CONNECT 함수 56

4.3 BIND 함수 58

4.4 LISTEN 함수 61
4.5 ACCEPT 함수 66
4.6 FORK 및 EXEC 함수 69
4.7 CONCURRENT 서버 72
4.8 CLOSE 함수 75
4.9 GETSOCKNAME 및 GETPEERNAME 함수 76

5. TCP CLIENT/SERVER 예제 77

5.1 TCP ECHO SERVER 78
5.2 TCP ECHO CLIENT 80
5.3 정상적인 시작과 종료 82
5.4 시그널 처리함수 87
5.5 비정상적인 연결 종료 100
5.6 SIGPIPE 시그널 103
5.7 ABNORMAL SERVER TERMINATION 105
5.8 TCP 예제 요약 107
5.9 USER DATA FORMAT 109

6. I/O MULTIPLEXING: SELECT() 113

6.1 I/O 모델 114
6.2 SELECT 함수 121
6.3 STR_CLI 함수: REVISITED 127
6.4 SHUTDOWN 함수 129
6.5 STR_CLI 함수: REVISITED AGAIN 131
6.6 TCP ECHO 서버: REVISITED 132
6.7 PSELECT 함수 135

7. SOCKET OPTIONS 137

7.1 GETSOCKOPT 및 SETSOCKOPT 함수 137
7.2 SOCKET OPTION 상태 검사하기 139
7.3 GENERIC SOCKET OPTIONS 144

7.4 IPv4 Socket Options 157

7.5 IPv6 Socket Options 159

7.6 TCP Socket Options 162

7.7 SCTP Socket Options 166

7.8 Fcntl 함수 179

8. UDP SOCKETS 182

8.1 recvfrom 및 sendto 함수 183

8.2 UDP Echo 서버 184

8.3 UDP Echo Client 187

8.4 Lost Datagram 189

8.5 Server Not Running 193

8.6 UDP 예제 요약 195

8.7 UDP를 사용하는 connect 함수 197

8.8 Flow Control이 없는 UDP 203

8.9 UDP에서 Outgoing Interface의 결정 206

9. SCTP SOCKETS 207

9.1 Interface 모델 208

9.2 sctp_bindx 및 sctp_bindx 함수 213

9.3 SCTP 주소처리 함수 216

9.4 sctp_sendmsg 함수 218

9.5 sctp_recvmsg 함수 219

9.6 sctp_opt_info 함수 219

9.7 sctp_peeloff 함수 220

9.8 shutdown 함수 221

9.9 Notifications 222

10. SCTP CLIENT/SERVER 예제 228

10.1 SCTP One-to-Many Style Streaming Echo Server 229

10.2 SCTP One-to-Many Style Streaming Echo Client 231

10.3 SCTP Streaming Echo Client 234

10.4 HoL(Head-of-Line) Blocking 236
10.5 Stream 수의 제어 244
10.6 세션 종료의 제어 245

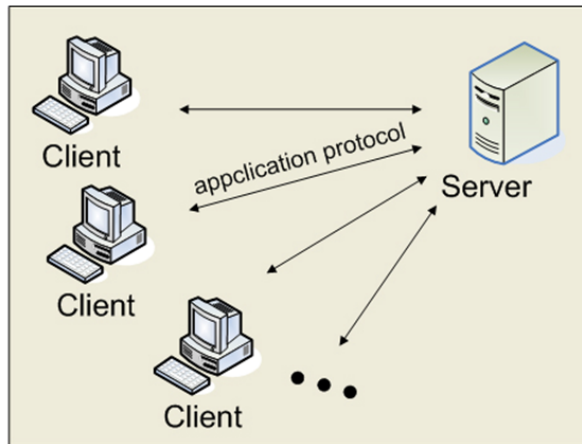
부록 A. TCP/IP 인터넷 프로토콜 247

A.1 인터넷 서비스 248
A.2 인터넷 주소와 도메인 이름 251
A.3 모바일 인터넷 255
A.4 다양한 인터넷 통신망 258
A.5 프로토콜 스택 262
A.6 Internet Protocol (IP) 263
A.7 User Datagram Protocol (UDP) 269
A.8 Transmission Control Protocol (TCP) 271
A.9 Stream Control Transmission Protocol (SCTP) 276
A.10 응용 계층 프로토콜 278

부록 B. LNP.H 헤더파일 282

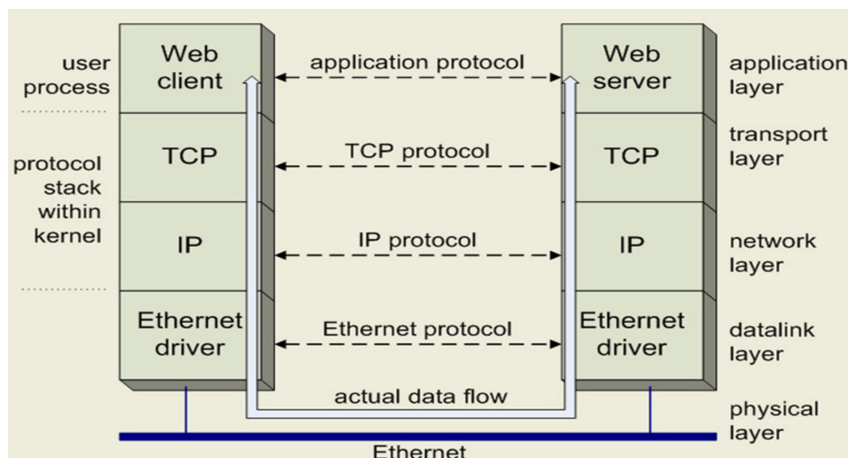
1. 서론

인터넷 응용 프로그램은 대부분 Client-Server 구조를 따른다. 서버에는 장시간 구동하는 데몬(daemon) 형태의 프로그램이 탑재되어 다른 Client의 접속을 기다린다. 반면에 Client는 필요할 때에 서버에 접속하여 원하는 서비스를 받는다. 그림 1.1처럼 대개 서버는 불특정 다수의 Client와 통신한다.



<그림 1.1> Client-Server 모델

Client 응용과 서버 응용은 네트웍 프로토콜을 통해서 통신하는데, 네트웍 프로토콜은 여러 계층(layer)의 복잡한 구조로 되어있다. 우리는 주로 TCP/IP 프로토콜을 다룬다. 예를 들어, 웹 Client와 웹서버는 TCP를 이용해 통신한다. TCP는 IP(Internet Protocol)와 통신하고, IP는 링크계층과 통신한다. 그림 1.2는 TCP/IP 프로토콜과 인터넷 통신과정을 보여준다.



<그림 1.2> 네트웍 프로토콜

1.1 DAYTIME CLIENT 예제

먼저 다음 예제를 통해 소켓관련 개념과 용어들을 소개한다. 다음 예제에서 Client가 서버와 TCP 연결을 구성하면, 서버는 현재 시간과 날짜를 사람이 읽을 수 있는 형태로 표현해준다.

```
<daytimetcpcli.c>
-----------------------------------------------------------------------------
1  #include "lnp.h"
2  int
3  main (int argc, char **argv)
4  {
5      int     sockfd, n;
6      char    revline[MAXLINE + 1];
7      struct sockaddr_in servaddr;

8      if (argc !=2)
9              err_quit ("usage: a.out <IP address>");
10     if ((sockfd = socket(AF_INET, SOCK_STREAM, 0))<0)
11             err_sys("socket error");

12     bzero (&servaddr, sizeof(servaddr));
13     servaddr.sin_family = AF_INET;
14     servaddr.sin_port = htons(13);    /*daytime server*/
15     if (inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0)
16             err_quit("inet_pton error for %s", argv[1]);

17     if (connect(sockfd, (SA *) &servaddr, sizeof (servaddr)) < 0)
18             err_sys("fputs error");
19     while ((n=read(sockfd, recvline, MAXLINE)) > 0) {
20             revline[n] = 0;           /* null terminate */
21             if (fput(recvline, stdout) == EDF)
22                     err_sys("fputs error");
23     }
24     if (n<0)
25             err_sys("read error");
26     exit(0);
27 }
-----------------------------------------------------------------------------
```

<그림 1.3> TCP Daytime Client

이것이 본 교재에서 사용할 소스코드의 기본 형태이다. 위 예제 프로그램을 직접 입력, 실행 및 수정해보면 네트웍 프로그래밍 실력 향상에 큰 도움이 될 것이다.

아래처럼 default로 설정된 a.out으로 컴파일하여 실행하면 다음과 같이 출력된다.

```
[linux] a.out 206.168.112.96            our input
Mon May 26 20:58:40 2003          the program's output
```

Header 포함

모든 예제 코드는 “lnp.h” 헤더파일을 사용하는데, 본 교재의 서문에서 명시한 웹사이트에서 해당 파일을 구할 수 있다. 이 header는 소켓 프로그래밍에 필요한 여러 가지 시스템 header와 상수 등을 정의하고 있다.

Command-line 인수

command-line 인수와 함께 main 함수를 정의한다.

TCP socket 생성

socket 함수는 Internet(AF_INET), stream(SOCK_STREAM) 타입의 socket을 만드는데 이것을 TCP socket이라 한다. 이 함수는 정수 값을 descriptor로서 반환시켜 주는데, 이는 다음 함수(connect, read)의 호출 시에 socket을 지정하는데 사용된다. if문은 socket 함수의 호출을 포함하고, sockfd에 반환 값을 할당하며, 그 값이 0보다 작은지 검사한다. ‘socket’이라는 용어의 여러 가지 의미로 사용되는데, socket() 함수는 sockets API라고 불린다.

socket 호출에 실패하게 되면 err_sys 함수를 호출해 프로그램을 종료한다. 이 때, error message와 그에 대한 설명을 출력하고 프로세스를 종료 시킨다. err_로 시작하는 함수들은 이러한 용도로 쓰인다.

Server의 IP 주소와 port 지정

Internet socket address 구조체 변수(serrvaddr)를 server의 IP 주소와 port번호로 채운다. 이를 위해, 먼저 구조체 변수를 bzero() 함수를 이용해 초기화하고, address family를 AF_INET으로 정한 뒤, port 번호를 13으로 (daytime server의 well-known port 번호임) 정하며, IP 주소를 command-line의 첫 번째 인수(argv[1])로 정한다. 이 때, IP 주소와 port 번호는 일정한 형식을 지녀야 하는데, 이를 위해 htons("host to network short")와 inet_pton("presentation to numeric") 함수를 사용한다.

Server 접속

TCP 소켓이 사용될 때, connect() 함수는 socket address 구조체의 두 번째 인수가 가리키는 server와 TCP 연결설정을 시도한다. Connect()의 세 번째 인수로 socket address 구조체의 길이가 사용되는데, 대개 sizeof operator를 사용하여 길이를 구한다.

Server 의 응답 출력

Server로부터 받은 응답 결과를 표시하기 위해 표준 입출력 함수인 fputs()를 사용한다. Server의 응답은 보통 26-byte의 string형식을 가진다.

```
Mon May 26 20:58:40 2003
```

일반적으로 TCP의 단일 segment를 통해 26 byte의 data가 반환된다. 그러나 data의 크기가 커지면 하나의 read() 함수를 써서 모든 data를 읽을 수 없다. 따라서, 예제에서처럼 TCP socket을 읽을 때는, read()를 loop안에 넣어서 read 함수가 0를 반환하거나(상대방이 연결을 해제하는 경우) 0보다 적은 값을 반환하는 경우(error)에 loop를 마치도록 하는 게 바람직하다.

프로그램 종료

exit() 함수는 프로그램을 종료한다. 프로세스가 종료될 때 열려 있던 모든 descriptor들이 닫히며 TCP socket도 닫힌다.

1.2 오류처리 함수와 WRAPPING FUNCTIONS

실제 프로그래밍 작업에서 오류가 발생했을 때 오류의 원인을 찾기 위한 오류처리 함수의 사용은 필수적이다. 앞선 예제에서 socket(), inet_pton(), connect(), read(), fputs() 등의 함수호출에서 오류가 발생할 수 있고, 대개 오류발생 시 err_quit(), err_sys() 함수 등을 사용하여 오류메시지를 출력하고 프로그램을 종료한다.

한편, 오류처리를 위해 wrapper 함수를 사용하여 프로그램을 좀 더 짧게 만들 수 있다. 예를 들어 socket() 함수를 래핑(wrapping)하는 Socket() 함수를 다음과 같이 사용할 수 있다.

```
sockfd = Socket (AF_INET, SOCK_STREAM, 0);
```

여기서 Socket() wrapper function의 내용은 다음과 같다.

```
-----------------------------------------------------------------------------
int
Socket (int family, int type, int protocol)
{
       int    n;
       if ( ( n = socket(family, type, protocol)) < 0)
              err_sys("socket error");
       return (n);
}
-----------------------------------------------------------------------------
```

본 교재에서 대문자로 시작하는 함수들은 대부분 wrapper 함수를 의미한다.

Unix/Linux 소켓 API 함수에서 error가 발생 했을 때, 전역변수 errno은 해당하는 error type을 나타내는 값으로 바뀌고, 함수는 보통 -1을 반환한다. err_sys() 함수는 errno의 값을 보고 그에 대응하는 오류 메시지를 출력한다(예를 들어, errno이 ETIMEOUT이면 "Connection timed out"을 의미함). errno의 값은 함수가 오류 발생시에만 의미를 지닌다. 오류가 없으면 그 값은 의미가 없다. 모든 오류타입은 대문자 'E'로 시작하는 이름의 상수값으로 정의되며 <sys/errno.h> 파일에 정의되어 있다.

1.3 DAYTIME SERVER 예제

그림 1.4는 앞서 소개한 Daytime Client와 함께 구동되는 TCP Daytime Server 예제이다.

Server의 well-known port를 socket에 bind

bind()를 호출하여 socket address 구조체를 채우고 server의 well-known port(daytime service의 경우 13)에 연결한다. Server의 host가 여러 가지 interface를 가지는 경우 IP 주소를 INADDR_ANY로 지정하여 어떤 interface에서도 server가 client의 접속을 허용하도록 한다.

Socket 을 listening socket 으로 바꾸기

listen()을 호출하면 socket은 listening socket으로 바뀌게 된다. 보통의 TCP server의 경우 socket, bind, listen의 3단계는 listening descriptor(예제에서 listenfd)에 의해 처리된다. 상수 LISTENNQ는 lnp.h 파일에 정의된 것이다. 이것은 client가 kernel에 접속하기 위해 listening descriptor의 queue상에 있을 수 있는 최대 숫자를 뜻한다.

```
<daytimetcpsrv.c>
---------------------------------------------------------------------------
1  #include   "lnp.h"
2  #include   <time.h>
3  int
4  main (int argc, char **argv)
5  {
6       int    listenfd, connfd;
7       struct sockaddr_in servaddr;
8       char   buff [MAXLINE];
9       time_t ticks;
10      listenfd = Socket (AF_INET, SOCK_STREAM, 0);
11      bzero (&servaddr, sizeof(servaddr));
12      servaddr.sin_family = AF_INET;
```

```
13     servaddr.sin_addr.s_addr = htonl (INADDR_ANY);
14     servaddr.sin_port = htons (13);   /* daytime server */
15     bind (listendfd, (SA *) &servaddr, sizeof (servaddr));
16     Listen (listendfd, LISTENQ);
17     for ( ; ; ) {
18             connfd = Accept (listenfd, (SA *) NULL, NULL);
19             ticks = time(NULL);
20             snprintf (buff, sizeof (buff), "%.24s\r\n", ctime(&ticks));
21             Write (connfd, buff, strlen (buff));

22             close (connfd);
23     }
24 }
-------------------------------------------------------------------------------
```

<그림 1.4> TCP Daytime Server

Client 의 접속 승인과 응답 보내기

일반적으로 server process는 accept()를 호출하고 나서 client 연결이 도착할 때까지 대기한다. TCP 연결설정은 three-way handshake 방식을 따르는데, 이 handshake가 끝나면 accept() 함수가 반환되고, connected descriptor라 불리는 새로운 descriptor가 생성된다. (이는 listening descriptor 와는 다른 것임). 새로운 connected descriptor는 서버가 새로운 client와 통신하는데 사용되고, listening socket은 서버가 계속해서 다른 client를 기다리는 데에 사용된다. 서버는 새로운 client 를 위해 상기의 과정을 무한히 반복하므로 17번 line처럼 무한 loop를 사용한다.

현재 시간과 날짜는 library 함수인 time()에 의해서 반환되는데, 이 때 1970년 1월 1일 0시 0분 0초 이후에 몇 초가 경과 되었는지를 Coordinated Universal Time(UTC) 형태의 정수 값으로 주어진다. 그리고 library 함수 ctime()에 의해 정수 값이 사람이 읽을 수 있는 형태로 바뀌게 된다.

Mon May 26 20:58:40 2003

연결 종료

Server는 close()를 호출하여 client와의 연결을 끊는다. 이 때 TCP 연결을 종료하는 프로토콜 절차가 시작된다. TCP FIN이 각 방향으로 보내지고, 각 FIN은 그에 대한 승인(ACK) 패킷을 보낸다.

그림 1.4의 서버 코드에서 server는 한 순간에 단 하나의 client만 처리하며 반복적으로 수행되기 때문에 "iterative" server라 부른다. 동시에 다수의 client를 관리하기 위한 "concurrent server"를 구현하는 방법에는 여러 가지가 있다. 간단한 방법으로 fork() 함수를 호출하여(4장) 각 client마다 child process를 만들어 주는 것이다. 다른 방법으로 fork() 함수 대신 "thread(쓰레드)"를 이용하거나 pre-fork를 통해 server 시작 시 미리 children process를 생성해 두는 방법 등이 있다.

다음은 Daytime Client와 Server 예제 코드의 실행화면이다.

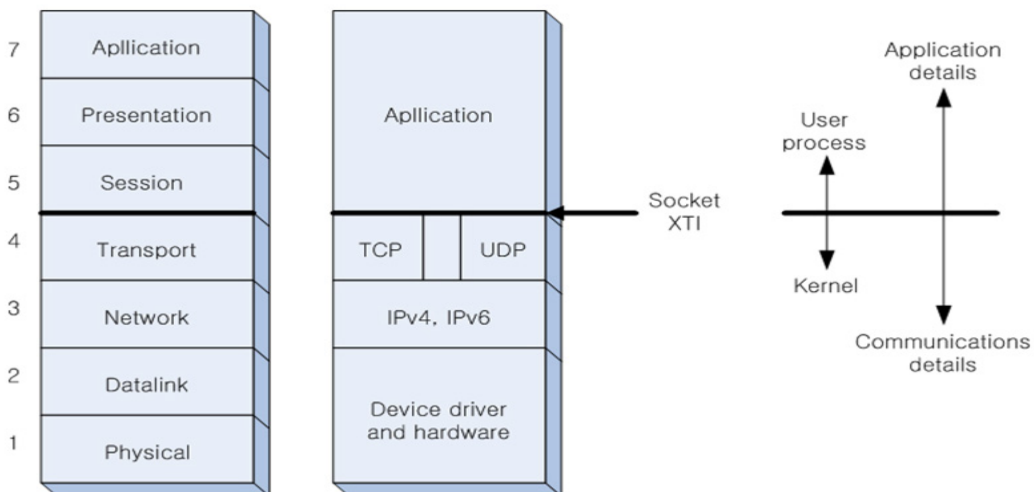
```
root@localhost:~/UNP/unpv13e
[root@localhost unpv13e]# ls
DISCLAIMER       daytimetcpcli.c    libfree    sig
Make.defines.in  daytimetcpcliv6    libgai     soc
Makefile.in      daytimetcpcliv6.c  libroute   soc
README           daytimetcpsrv      mcast      spa
VERSION          daytimetcpsrv.c    mysdr      ssn
aclocal.m4       debug              names      str
advio            icmpd              nonblock   tcp
bcast            inetd              oob        tes
config.guess     install-sh         ping       thr
config.h.in      intro              route      tra
config.sub       ioctl              rtt        udp
configure        ipopts             sctp       udp
configure.in     key                select     uni
daytimetcpcli    lib                server     unp
[root@localhost unpv13e]# cls
-bash: cls: command not found
[root@localhost unpv13e]#
[root@localhost unpv13e]#
[root@localhost unpv13e]#
[root@localhost unpv13e]#
[root@localhost unpv13e]#
[root@localhost unpv13e]#
[root@localhost unpv13e]#
[root@localhost unpv13e]#
[root@localhost unpv13e]# ./daytimetcpsrv

[영어 ][완성 ][두벌식 ]
```

```
root@localhost:~/UNP/unpv13e
[root@localhost root]# cd
[root@localhost root]# cd UNP
[root@localhost UNP]# cd unpv13e
[root@localhost unpv13e]# ./daytimetcpcli 155.230.105.16
1
Mon Dec 10 18:52:55 2007
[root@localhost unpv13e]#

[영어 ][완성 ][두벌식 ]
```

<그림 1.5> daytimetcpcli.c와 daytimetcpsrv.c의 실행화면

1.4 TCP/IP 프로토콜과 소켓 프로그래밍

컴퓨터통신 프로토콜들은 OSI(Open System Interconnection) model을 따른다. 이 model은 7개의 계층(layer)으로 구성되어 있으며, 그림 1.6에 Internet protocol suite와 각 layer가 대응되는 부분을 비교해 놓았다.



<그림 1.6> OSI 모델, TCP/IP 프로토콜과 소켓 프로그래밍

OSI layer의 가장 아래 2 layer는 system에 공급되는 device driver와 networking hardware이다. 우리는 주로 1500 바이트(byte)의 MTU(maximum transfer unit)를 갖는 이더넷(Ethernet)을 data link layer로 고려할 것이다.

Network layer로는 IPv4와 IPv6 프로토콜이 있다. Transport layer는 2장에서 다룰 TCP와 UDP가 담당한다 (혹은 9장과 10장에서 다루는 SCTP도 이에 해당한다). 그림 1.6에서 TCP와 UDP사이에 약간의 틈을 두었는데 이는 application에서 transport layer를 거치지 않고 바로 IPv4, IPv6를 이용할 수 있음을 뜻한다. 이를 raw socket이라 한다 (ICMP도 raw socket을 이용하여 구현된다).

OSI model의 상위 3 layer를 통틀어 application layer라 한다. Web client (browser), Telnet client, Web server, FTP server 등의 application이 이에 속한다. Internet protocol에서는 OSI model의 상위 3 layer에 뚜렷한 구분이 없다.

Socket programming interface는 상위 application에서 transport layer로 통하는 interface 나타낸다. 이것이 소켓 프로그래밍의 핵심 내용이다 (TCP나 UDP socket을 이용하여 application프로그램을 작성한다). Raw socket의 사용법은 본 교재에서 다루지 않는다.

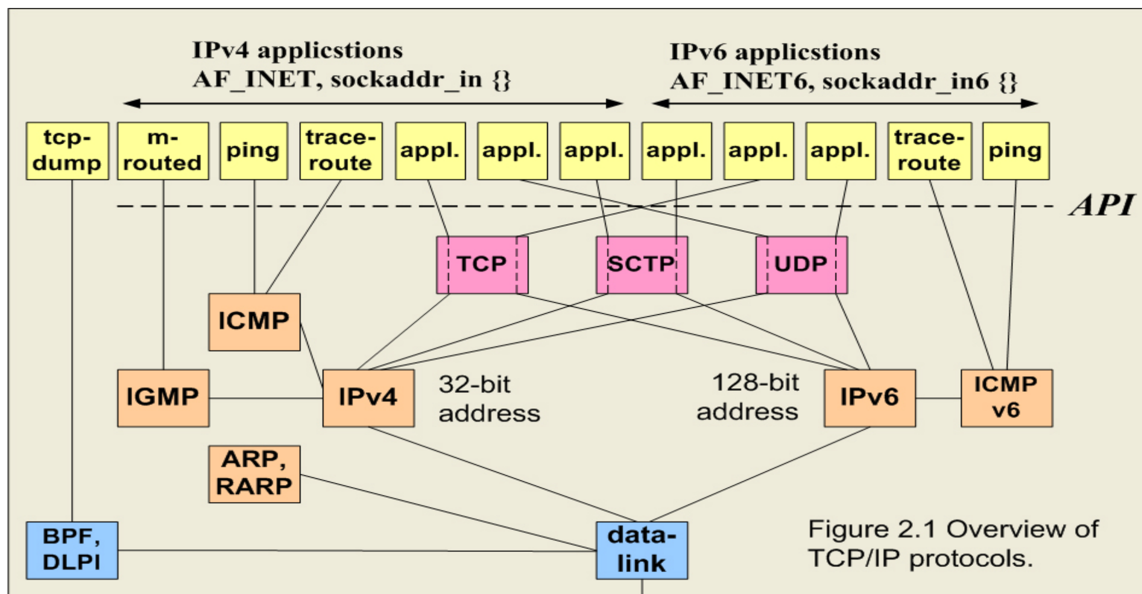
그림 1.6에서 한가지 주목할 사항이 있다. 바로 상위 응용계층에서 transport layer로 통하는 인터페이스(interface)로서 socket API가 제공된다는 점이다. 상위 application (예: FTP, Telnet, HTTP) 프로토콜은 주로 사용자의 서비스 측면을 다루는 반면에 하위 4개의 계층은 응용과는 무관하게 데이터 통신을 관리하는 기능을 수행한다 (예: data 전송, 오류제어, 흐름제어 등). 즉, 소켓 프로그래밍에서는 하위의 TCP/IP 프로토콜의 세부 동작내용에 대한 깊은 지식이 없이도 응용 프로그램 개발이 가능하다 (이를 자료의 "추상화(abstraction)" 기법이라 한다). 대개 소켓 API의 상위 응용계층은 user process로 구현되고 하위 4개의 계층은 OS(operating system)의 커널(kernel)에서 제공된다. Unix/Linux나 다른 OS들도 user process와 kernel을 구분 짓는다. 이러한 의미에서 "socket API"라는 표현을 사용한다.

2. 수송계층 프로토콜: TCP, UDP, SCTP

이 장에서는 예제를 통해 TCP/IP 프로토콜 스택의 수송계층 프로토콜에 해당하는 TCP, UDP, SCTP에 대하여 살펴본다. 대부분의 client/server application은 TCP나 UDP를 사용하고 있다. SCTP는 본래 인터넷 전화 혹은 VoIP(Voice over IP) 서비스의 신호전송(signaling) 등의 "mission-critical" 인터넷 응용을 위해 개발된 새로운 수송계층 프로토콜이다. 하지만 TCP, UDP처럼 일반적인 응용 프로그램도 SCTP를 이용하여 개발할 수 있다.

UDP는 간단하고 신뢰성(reliability) 제공이 필요 없는 응용에 사용되고, TCP는 신뢰전송이 요구되는 응용을 위해 사용되며 byte stream 기반의 protocol이다. SCTP는 신뢰성이 있다는 점에서 TCP와 유사하지만 바이트 기반이 아닌 메시지(message) 기반의 프로토콜이며 transport level에서의 multi-homing, multi-streaming 등의 새로운 특징을 제공한다. 상세한 프로토콜 내용은 "컴퓨터 네트워크" 관련 강의에서 다루며, 소켓 프로그래밍 관점에서는 해당 소켓에서 어떤 transport protocol을 사용하는 지만 인식하면 된다. 하부 프로토콜에 따라 소켓의 기능 및 동작이 달라질 것이다.

먼저, 인터넷통신을 위한 TCP/IP 프로토콜 스택에는 TCP, IP 이외에도 많은 프로토콜이 있다. 그림 2.1은 TCP/IP 프로토콜들의 상호 관련성을 보여준다.



<그림 2.1> TCP/IP 프로토콜

위 그림에는 IPv4와 IPv6 모두가 포함되어 있다. 소켓 프로그래밍에서 IPv4와 IPv6를 구별하기 위하여 AF_INET 혹은 AF_INET6 상수가 사용된다. 가장 왼쪽의 application인 tcpdump는 BSD packet filter(BPF)나 datalink provider interface(DLPI)를 써서 datalink와 바로 통신하며 주로 패킷 감시 및 통계자료를 추출하기 위해 사용된다. 점선 아래 9개의 프로토콜들은 API 형태로 제공된다. 그림에서 traceroute program이 IP와 ICMP 두 socket을 사용 한다.

그림 2.1에서 언급된 주요 프로토콜들에 대하여 간략히 살펴보면 다음과 같다.

IPv4	Internet Protocol version 4. 32-bit 주소를 가진다. IPv4 는 TCP, UDP, SCTP, ICMP, IGMP 의 packet 수송 service 를 제공한다.
IPv6	Internet Protocol version 6. IPv6 는 1990 년대 중반에 IPv4 를 대체하기 위해 고안되었다. 1990 년대의 폭발적인 Internet 사용 증가로 주소가 128-bit 로 늘어났다. IPv6 도 TCP, UDP, SCTP, ICMPv6 의 packet 수송에 service 를 제공한다. 우리가 흔히 사용하는 "IP"라는 용어는 IP 계층과 IP 주소를 의미하고 이 경우, IPv4 와 IPv6 를 구별 할 필요는 없다.
TCP	Transmission Control Protocol. TCP 는 연결 지향적인 프로토콜이며 신뢰성을 제공하고 양방향(full-duplex) byte stream 기반이다. TCP socket 은 stream socket 의 일종이다. TCP 는 오류제어, 흐름제어 및 혼잡제어 기능을 제공한다. 대부분의 인터넷 응용은 TCP 를 사용한다. TCP 는 IPv4 나 IPv6 를 이용한다.
UDP	User Datagram Protocol. UDP 는 connectionless protocol 이며 datagram socket 의 일종이다. UDP 에서는 오류제어 및 오류복구 기능을 수행하지 않는다. TCP 나 UDP 모두 IPv4 나 IPv6 에서 사용 할 수 있다.
SCTP	Stream Control Transmission Protocol. TCP 와 유사하나, multi-homing 및 multi-streaming 등의 고유 특성을 제공. TCP, UDP, SCTP 모두 IPv4 나 IPv6 를 사용.
ICMP	Internet Control Message Protocol. ICMP 는 IP 패킷 전송에 대한 제어기능을 제공함. ping 이나 traceroute 등의 프로그램은 ICMP 를 사용. 이러한 message 들은 보통 user process 가 아닌 TCP/IP networking software 에 의해 스스로 만들어 진다. ICMPv6 과 구별하려고 ICMPv4 라고 쓰기도 한다.
IGMP	Internet Group Management Protocol. IPv4 의 multicasting 에 쓰인다.
ARP	Address Resolution Protocol. ARP 는 IPv4 주소를 hardware 주소(혹은 Ethernet 주소)로 바꾸어 준다. ARP 는 보통 Ethernet, token ring, FDDI 같은 broadcast network 에 쓰이고 point-to-point network 에서는 쓰이지 않는다.

ICMPv6	Internet Control Message Protocol version 6 의 약자. ICMPv4, IGMP, ARP 의 기능적인 측면들을 통합.
BPF	BSD Packet Filter 의 약자. datalink layer 로의 접근을 제공하며 패킷감시 및 데이터전송 관련 통계자료 등을 얻기 위해 사용됨.

2.1 USER DATAGRAM PROTOCOL (UDP)

UDP는 간단한 transport-layer protocol이다. 응용(application)에서 UDP socket에 message를 작성하면 UDP datagram으로 캡슐화 되었다가 IP datagram으로 캡슐화 되어 목적지로 보내진다. 그러나 UDP는 오류제어 기능이 없으므로 datagram이 network를 통해 원하는 목적지까지 한번에 도달한다는 보장이 없다. 따라서, 만약 UDP datagram이 최종 목적지까지 전송되는 것을 보장하려면, checksum, error 검출, 자동 재전송 등의 기능을 응용 프로그램에서 직접 제공해야 한다.

또한 UDP가 비연결형 service를 제공한다고 했는데, 이는 UDP client와 server간의 장시간에 걸친 통신이 없기 때문이다. 예를 들어, UDP client가 socket을 생성하여 datagram으로 보내고 나서 즉각적으로 같은 소켓을 사용하여 다른 datagram을 보내는 것도 가능하다. 마찬가지로 UDP server 또한 다수의 UDP client에서 오는 다수의 datagram을 하나의 UDP socket으로 받을 수 있다.

2.2 TRANSMISSION CONTROL PROTOCOL (TCP)

TCP가 application에 제공하는 service는 UDP가 제공하는 service와는 다르다. TCP는 client와 server 사이의 연결(connection)을 제공한다. TCP client는 주어진 server와 연결하고 서로 data를 주고 받으며 통신이 끝나면 연결을 종료한다.

TCP는 신뢰전송 기능을 제공한다. TCP에서 data를 다른 곳에 전송하려 할 때에는 상대방에게 ACK(acknowledgment) 응답을 요구한다. 만약 ACK가 도착하지 않으면 TCP는 자동적으로 data를 재전송하고 더 긴 시간 동안 대기한다. 이후 몇 번의 재시도에도 성공하지 못하면 포기하게 되는데 이때까지 걸리는 시간은 보통 4~10분 정도 이다 (network에 따라 차이가 있다). TCP라고 해서 100% 신뢰할 수 있는 것은 아닌 것이다. 예를 들어, 상대방 endpoint가 아예 data를 받을 수 없는 경우에는 (예: 인터넷 연결이 끊긴 경우) 전송을 보장 할 수 없다. 전송에 실패했을 경우에 TCP는 실패한 사실을 상위 응용에 알려준다.

TCP는 client와 server사이에 RTT(Round Trip Time) 시간을 측정하는 기능을 가지고 있다. RTT는 network traffic에 영향을 받으며 TCP의 전송성능에 큰 영향을 주므로 TCP 전송 도중에 RTT를 지속적으로 측정할 필요가 있다.

TCP는 오류복구(error recovery) 기능을 제공한다. 전송하는 모든 데이터에 byte 번호를 매겨 data의 순서를 정한다. 예를 들어 application이 2048 byte를 TCP socket에 썼다고 가정하면 TCP는 두 개의 segment를 보내게 되는데 첫 번째 segment는 1~1024번까지의 data를 가지고 두 번째 segment는 1025~2048번까지의 data를 가진다(segment는 TCP가 IP 전송하는 data의 단위이다.). 만약 data들의 순서가 뒤바뀌어 전송이 되더라 하더라도 앞서 지정해 놓은 순번(sequence number)이 있어서 바로잡는 것이 가능하다 (이를 reordering이라 한다). 중복으로 같은 data를 받은 경우(전달이 잘 되었어도 송신측이 이를 인지하지 못하고 재전송을 하는 경우)에도 순번에 의거하여 나머지 하나는 폐기(discard) 할 수 있다. 반면에, UDP는 신뢰할 수 없다. UDP는 ACK, 순번, RTT, timeout, 재전송 등의 기능을 제공하지 않는다. 예를 들어, 만약 UDP datagram이 network상에서 중복된다면 수신측 host는 두 개의 data를 받게 된다.

TCP는 또한 flow control을 제공한다. TCP는 항상 상대방에게서 받을 수 있는 data의 byte수를 알려준다. 이것을 advertised window라 부른다. 어떤 때이든 window는 현재 수신 가능한 buffer의 크기를 나타내며 sender에서 overflow가 일어나는 것을 방지한다. 시간이 지남에 따라 window는 유동적으로 변한다. Sender로부터 data를 받았을 때는 window의 크기가 줄어들고 수신측 application이 buffer로부터 data를 읽어 오면 window의 크기는 증가한다. window의 크기가 0이 되는 경우도 있다. TCP의 수신 buffer가 소켓에 대해 모두 소진된 상태이며 상대방으로부터 data를 받기 위해서는 application이 수신 buffer을 읽을 때까지 기다려야 한다. 반면에, UDP는 flow control을 제공하지 않는다. 빠른 UDP sender는 UDP receiver가 다 받지 못할 정도로 빠르게 datagram을 보내기 쉽다.

마지막으로 TCP의 연결은 양방향(full-duplex) 전송을 제공한다. 이는 application이 어느 때이든 송수신을 양방향으로 할 수 있다는 것을 의미한다. 또 TCP는 sender, receiver 각각에 대해 순번이나 window크기 같은 상태정보를 따라가야 한다. UDP도 양방향 전송이 가능하다.

2.3 STREAM CONTROL TRANSMISSION PROTOCOL (SCTP)

SCTP는 UDP, TCP와 유사한 service를 제공한다. SCTP는 RFC 2960에 기술되었고, RFC 3309에서 update 되었다. SCTP는 client와 server 사이에 association을 제공한다 (이는 TCP connection과 유사한 용어이다). SCTP는 TCP처럼 신뢰성, 순차적, flow control, 양방향 data 교환 기능을 application에게 제공한다. SCTP에서는 '연결(connection)' 이라는 용어 대신 'association'을 사용하는데 이는 SCTP multi-homing 특성을 강조하기 위해서 등장한 용어이다. SCTP는 두 system 사이에도 2개 이상의 IP를 사용할 수 있다.

TCP와는 달리 SCTP는 message-oriented이다. 즉, TCP의 바이트 기반 전송이 아니라 UDP처럼 메시지 단위의 전송 기능을 제공한다. SCTP는 응용 데이터의 메시지들을 여러 개로 구분하여 multiple stream으로 전송할 수 있는 multi-streaming 기능을 제공한다. 이 stream중 하나에서 message의 손실이 발생해도 다른 stream의 전달에는 영향을 주지 않는다. 이는 단일 stream 만을 전송하여 어느 한 data의 손실이 발생하면 손실된 데이터가 복구될 때까지 모든 데이터 전달을 중단시키는 TCP와는 대조적인 특성이다.

SCTP는 하나의 SCTP endpoint에 다수의 IP 주소를 사용하는 multi-homing 기능도 제공한다. 이는 network 실패에 강인한 network를 구성한다. SCTP는 한 경로에서 실패가(손실이) 발생하면 SCTP association의 다른 경로를 통해 데이터 전송을 계속할 수 있다.

2.4 TCP 연결 설정 및 종료

TCP의 connect(), accept(), close() 함수의 이해를 돕기 위해서 TCP 연결의 설정과 종료 그리고 TCP 상태천이도(state transition diagram)에 대해서 이해할 필요가 있다.

Three-Way Handshake

TCP 연결 설정은 다음과 같은 scenario에 의하여 이루어진다.

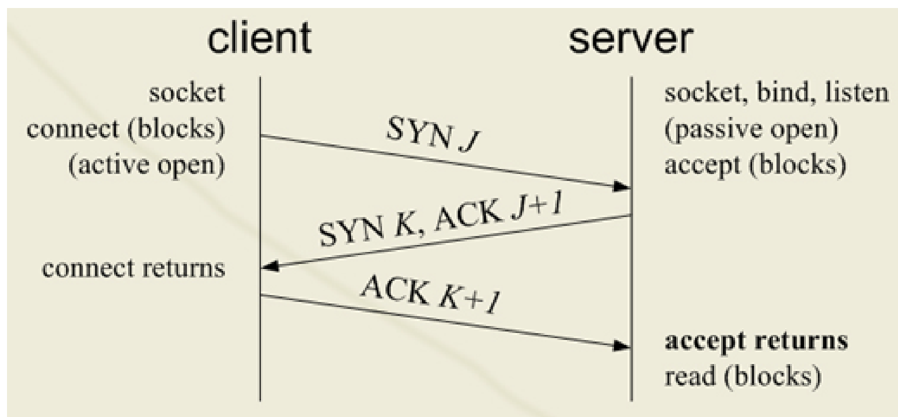
1) Server는 들어오는 연결을 받을 수 있도록 준비해야 한다. 이것은 보통 socket, bind, listen을 호출하면 되는데 passive open이라 부른다.

2) Client는 connect()를 호출하여 active open을 수행한다. 이 방법은 TCP client가 이 연결을 통해 보낼 data에 대한 초기 순번을 server에게 알려주는 "synchronize"(SYN) segment를 보내주는 것이다. 보통 SYN은 응용 데이터는 포함하지 않으며 프로토콜과 관련된 IP header, TCP header 등으로 구성된다.

3) Server는 client의 SYN에 대한 ACK를 보내면서 자신이 보낼 data의 초기 순번을 담아 보낸다. Server는 SYN segment를 보내면서 client의 SYN에 대한 ACK도 포함시킨다.

4) Client는 sever의 SYN에 대한 ACK를 보냄으로써 연결설정이 종료된다.

이러한 연결설정에 최소 3개 이상의 packet 교환이 필요하다고 하여 이 방법을 TCP의 three-way handshake라고 부른다. 그림 2.2는 이러한 절차를 보여준다.



<그림 2.2> TCP 연결설정 절차

그림에서 Client의 초기 순번을 J라 하고 server의 초기 순번을 K라 하겠다. ACK에 있는 순번은 다음에 받을 data의 순번이다. SYN에 순번은 1byte를 차지하며 ACK의 acknowledgment 번호는 각 SYN의 초기 순번에 1을 더한 것이다.

TCP Connection Termination

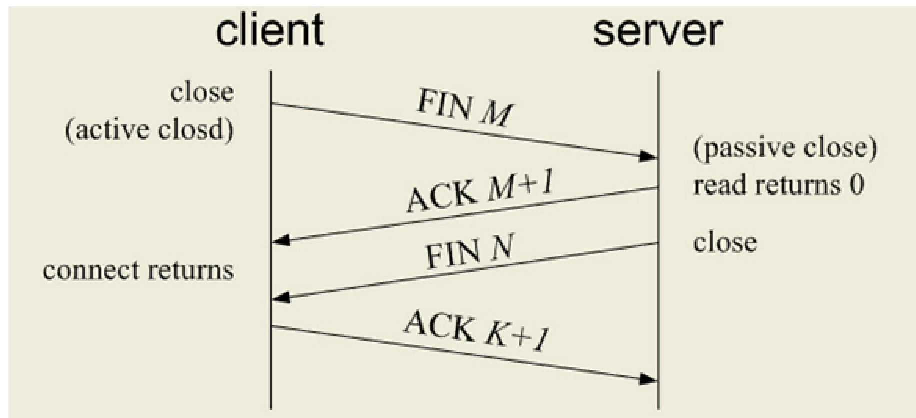
그림 2.3에서 보이듯이 연결 설정에 3개의 segment가 쓰이는 반면, 연결의 종료에는 4개의 segment가 쓰인다.

1) 먼저 한쪽의 application이 close를 호출하는데 이를 active close라 한다. 이 TCP는 종료를 의미하는 FIN segment를 보낸다.

2) 반대쪽에서는 FIN을 받아서 passive close를 수행한다. TCP는 받은 FIN에 대한 ACK를 보낸다. FIN의 수신은 응용이 연결을 통해서 더 이상 data를 받지 않음을 의미한다.

3) 그 후에 application은 socket을 close한다. 이어서 TCP는 FIN을 보낸다.

4) TCP가 마지막 FIN을 받으면(active close를 수행한 쪽) FIN의 ACK를 보내준다.



<그림 2.3> TCP 연결종료 절차

각 방향으로 FIN과 ACK가 필요하므로 일반적으로 4개의 segment가 요구된다. '일반적으로'라고 한 이유는 첫 번째 단계에서 FIN을 data와 함께 보내는 경우도 있기 때문이다. 2, 3 단계의 segment는 passive close를 한 쪽에서 나오는데 하나의 segment로 묶을 수도 있다.

SYN과 마찬가지로 FIN도 1-byte의 순번(sequence number)을 소모한다. 그래서 각 FIN의 ACK의 순번은 FIN의 순번에 1을 더한 것과 같다.

2단계와 3단계 사이에 passive close를 한 쪽에서 active close를 한 쪽으로 data가 흘러갈 가능성이 있다. 이를 "half-close"라 하며 6장의 shutdown 함수에서 자세히 설명하기로 한다.

그림 2.3에서 client가 active close를 하는 것으로 나오지만 server도 active close를 수행할 수 있다. 주로 client가 active close를 하지만 HTTP 같은 protocol에서는 server가 수행한다.

TCP State Transition Diagram

연결 설정과 연결 종료에서의 TCP 동작은 그림 2.4와 같은 상태천이도(state transition diagram)로 나타낼 수 있다. 하나의 연결에 대해서 11개의 서로 다른 상태(state)를 정의하는데 상태 전이는 현재 상태에서 수신한 segment의 종류에 의해 결정된다. 예를 들어, application이 CLOSED 상태에서 active open을 하면 TCP는 SYN을 보내게 되고 상태는 SYN_SENT로 바뀌게 된다. 만약 TCP가 SYN과 ACK를 받으면 ACK를 보내고 상태를 ESTABLISHED로 바꾼다. 이 최종 상태에서 대부분의 data 전송이 일어난다.

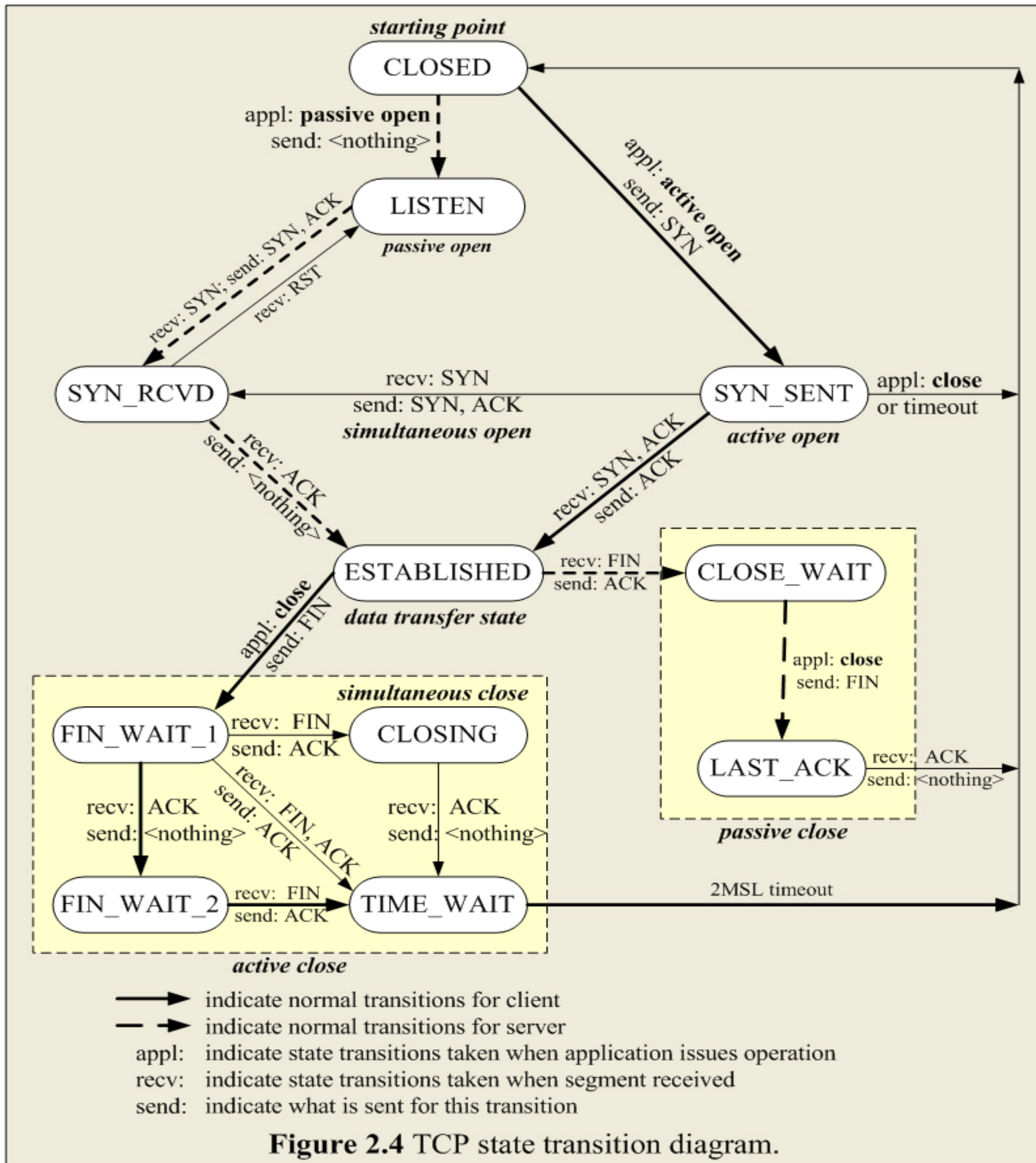


Figure 2.4 TCP state transition diagram.

<그림 2.4> TCP 상태천이도

ESTABLISHED 상태에서 나오는 두 개의 화살표는 연결 종료와 관계가 있다. Application이 파일끝(EOF: end of file) 신호를 받기 전에 close를 호출하면(active close), FIN_WAIT_1 상태로 옮긴다. 그러나 application이 ESTABLISHED 상태에서 FIN을 받게 되면(passive close) CLOSE_WAIT 상태로 옮기게 된다. 보통 client의 전이는 굵은 선으로 표시하고 sever의 전이는 굵은 점선으로 표시한다. 동시 열기(simultaneous open)와 동시 닫기(simultaneous close, 양쪽이 같은 시간에 FIN을 보낼 때)는 여기서 다루지 않겠다.

TCP 연결의 현재 상태들은 netstat 명령어를 이용하여 볼 수 있으며 5장에서 언급한다.

Watching the Packet

그림 2.5에 연결설정, data 전달, 연결종료에 대한 TCP packet 교환을 보여준다. 이때, 각 endpoint의 TCP 상태도 보여준다.

그림에서 Client는 MSS를 536 바이트임을 알려주고 server는 MSS를 1460 (Ethernet의 MTU에 대응되는 값)임을 알려준다. 각 방향에 대해 MSS는 달라도 무관하다.

한번 연결이 설정되면 client는 데이터를 server에 보낸다. 우리는 이 데이터가 하나의 segment에 모두 포함된다고(즉, server 알려준 1460보다 작은) 가정하겠다. Server는 이 데이터를 처리하고 응답 패킷을 보내는데 이 또한 하나의 segment에 들어간다고(이 경우 536보다 작다) 가정한다. Data segment는 굵은 화살표로 표시한다. 서버는 Client의 데이터에 대한 ACK를 함께 보낸다. 이를 "piggybacking"이라 부르며, server가 자신이 보낼 데이터 패킷을 구성하는데 소요되는 시간이 200ms보다 짧으면 ACK를 piggybacking하게 된다. 만약 시간이 더 오래 소요되면 ACK 패킷을 먼저 보내고 나중에 자신의 데이터를 보내게 된다.

연결을 종료시키는 4개의 segment를 살펴보면, active close를 수행하는 쪽(예제에서 client)이 TIME_WAIT 상태가 된다. 이 점에 대해서는 다음 장에서 논의 한다.

그림 2.5에서처럼 만약 TCP 연결을 통해 단지 하나의 데이터 segment를 보내고 하나의 데이터 segment를 답장으로 받는 것이라면, TCP는 총 10개의 segment가 발생한다는 것을 주목하자. 만약 UDP를 사용한다면 단 2개의 데이터 segment만 교화하면 된다 (연결 설정과 종료 과정에 없으므로). 그러나 UDP에서는 TCP가 제공하는 신뢰성 기능이 없고 이러한 기능을 UDP application에서 구현해야 한다. TCP는 혼잡제어(congestion control) 기능도 제공 하는데 이 또한 UDP application이 처리해야 한다. 그럼에도 불구하고, 많은 application은 적은 양의 data를 교환하면 되는 UDP를 이용하여 구현되기도 한다.

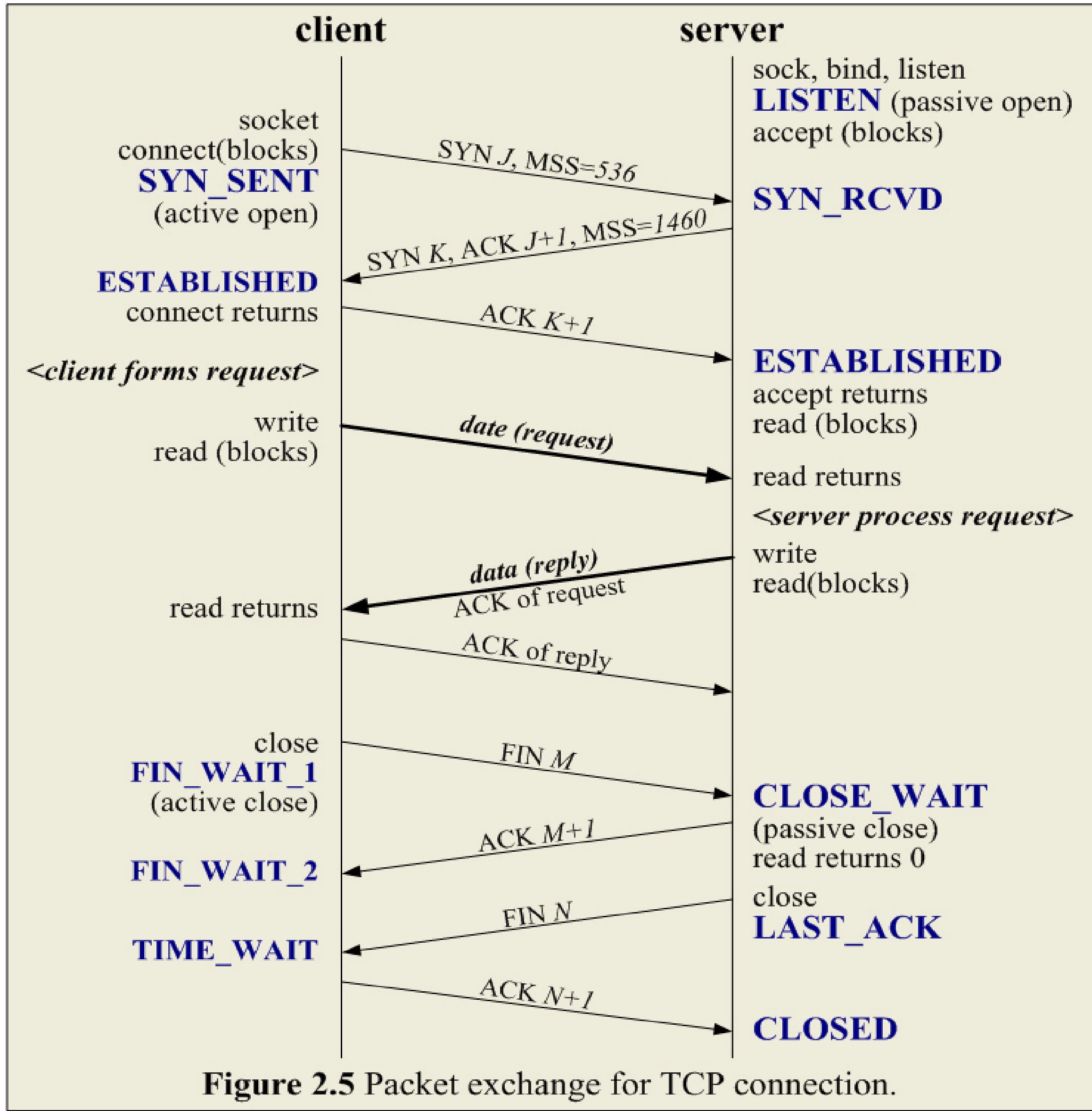


Figure 2.5 Packet exchange for TCP connection.

<그림 2.5> TCP 연결에서의 패킷 교환

2.5 TIME_WAIT 상태

TCP 기반 network programming에 있어 가장 오해가 많은 부분 중 하나가 TIME_WAIT 상태이다. 그림 2.4에서 active close를 하는 쪽이 이 상태를 거치는 것을 볼 수 있다. 이 상태에서 머무는 시간은 maximum segment lifetime(MSL)의 두 배이며, 2MSL이라 부르기도 한다.

모든 TCP 구현은 MSL 값을 지정하고 있다. RFC1122에서는 2분을 추천하지만, Berkeley에서 파생된 구현은 전통적으로 30초를 사용한다. 이는 TIME_WAIT 상태에서 머무는 시간이 1분에서 4분

사이임을 의미한다. MSL은 "어떤 IP datagram이 network상에서 생존할 수 있는 최대 시간"이다. 소켓 프로그래밍에 의해 생성된 모든 packet은 Internet에서 MSL보다 길게 유지될 수는 없다.

Packet을 네트웍 상에서 잃어버리는 것은 보통 변칙적인 routing의 결과이다. Router가 죽거나 두 개의 router의 연결이 끊어지면 routing protocol은 새로운 경로를 찾는데 수분 수초의 시간이 걸린다. 이 시간 동안 routing loop이 생길 수 있으며 이 loop에 packet이 갇힐 수 있다. 그러는 동안 TCP segment는 없어진 것으로 간주 되고 송신측은 재전송을 하여, 최종 목적지에 도달하게 된다. 그러나 이후에(사라진 packet의 MSL안에) routing loop이 고쳐져서 갇혀 있던 packet이 최종 목적지로 다시 도달할 수 있다. 이 경우 이 packet을 "lost duplicate" 또는 "wandering duplicate"이라 부른다. TCP는 이러한 복사본들을 관리 해야만 한다.

여기에 TIME_WAIT 상태를 두는 두 가지 이유가 있다.

1. 신뢰적인 양방향(full-duplex) 연결 종료의 구현
2. Network상의 오래된 복사본 segment를 소멸

첫 번째 이유는 그림 2.5에서 client가 보낸 마지막 ACK가 네트워크에서 손실되었다고 생각해보자. Server는 마지막 FIN을 다시 보낸 다음 client로부터 마지막 ACK를 받지 못했으므로 일정 시간 후에 다시 FIN을 client에게 보낼 것이다. 이러한 상황에서 만약 client가 TIME_WAIT 정보를 가지고 있지 않다면, client는 server의 FIN 메시지를 이해할 수가 없고, RST(reset) 패킷으로 서버에 응답하게 될 것이다. TCP의 연결종료 과정에서 4개의 segment 중에서 어느 것이 소실 되더라도 바르게 처리할 수 있어야 한다. 이러한 목적으로 TCP에서는 active close를 하는 쪽에서 TIME_WAIT 상태를 관리하고 마지막 ACK를 관리하도록 하고 있다.

TIME_WAIT 상태를 두는 두 번째 이유를 설명하기 위해서, 1.1.1.1 호스트의 port 1500번과 2.2.2.2 호스트의 포트 21번 사이의 TCP 연결을 가정한다. 이 연결이 닫히고 나서 잠시 뒤 같은 IP 주소와 port로 다른 연결이 설정된다고 생각해보자. IP 주소와 port 번호가 같으므로 첫 번째 연결의 "재생본(incarnation)"이라 부른다. TCP는 첫번째 연결에서 생성된 패킷이 나중에 도착하여 "재생본" 연결의 패킷으로 오인되는 것을 막아야 한다. 이를 위해 TCP는 TIME_WAIT 상태에 있을 때에는 새로운 "재생본(같은 port 사용) 연결을 발생시키지 못하도록 하고 있다. 이런 규칙을 적용함으로써 모든 TCP 연결은 이전 연결에서 생성된 패킷이 도착할 염려를 하지 않아도 된다.

2.6 SCTP ASSOCIATION 설정 및 종료

SCTP는 TCP처럼 connection-oriented이기 때문에 association의 설정과 종료절차를 수행해야 한다. 그러나 SCTP의 handshake는 TCP와는 다소 다르다.

Four-Way Handshake

TCP의 경우처럼 다음 scenario를 따라 SCTP association 설정이 이루어진다. 먼저, Server는 임의의client로부터의 접속요청을 승인할 준비를 해야 한다. 이 준비과정은 보통 socket, bind, listen을 호출하고 passive open을 하여 완료된다.

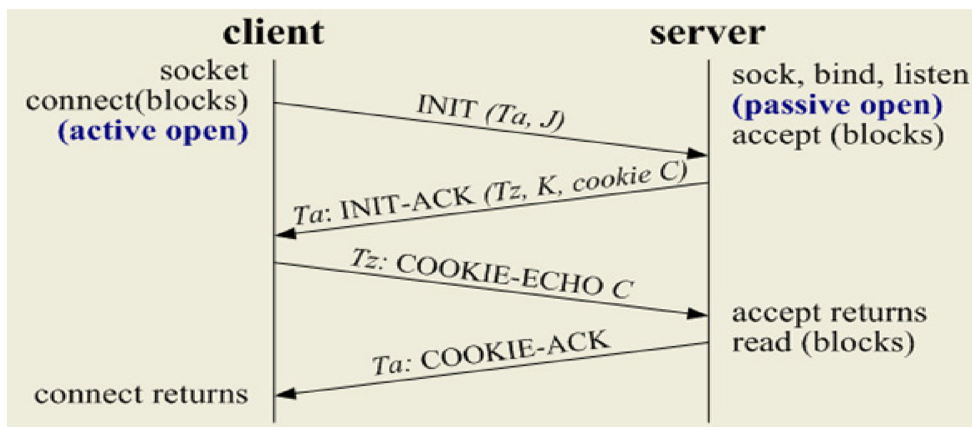
1) Client는 connect를 호출함으로써 active open을 수행한다. 이에 client SCTP는 INIT 초기화 message를 server에게 보내 IP 주소, 초기 sequence number 및 association 설정에 필요한 정보를 서버에게 알려준다.

2) Server는 서버의 IP 주소, 초기 순번, initiation tag 등의 정보, 그리고 state cookie 정보를 INIT-ACK message를 통해 client에게 보낸다. 상태 cookie는 숫자형태로 표시되며 다음 COOKIE-ECHO 메시지에 사용된다.

3) Client는 server의 상태 cookie에 대한 응답으로 COOKIE-ECHO message를 보낸다. 이 message는 packet안에 user data를 포함시킬 수 있다.

4) Server가 cookie의 정상적인 도착을 탐지하고 client에게 COOKIE-ACK message로 응답하면 association이 설정된다. 이 message는 packet안에 user data를 포함시킬 수 있다.

여기서 최소한 4번의 packet 교환이 필요한데 이 과정을 SCTP의 four-way handshake라 부른다. 그림 2.6을 통해 4 segment를 살펴볼 수 있다.



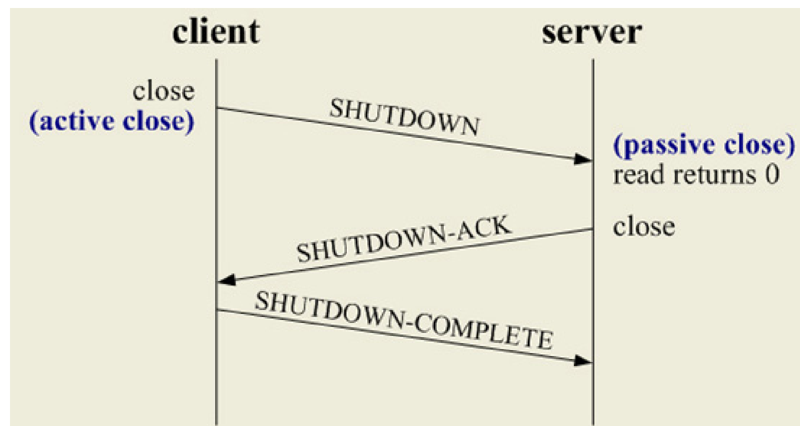
<그림 2.6> SCTP 연결 설정

SCTP의 four-way handshake는 cookie의 생성을 제외하고는 여러모로 TCP의 three-way handshake와 유사하다. INIT 패킷(혹은 chunk)는 initiation tag(Ta) 그리고 initial sequence number(J)를 담아 서버에게 보낸다. Ta는 추후 SCTP association에 대한 인증을 위해 사용되며 상대방에게 전송하는 모든 packet에게 포함되어야 한다 (SCTP 패킷헤더에 verification tag 항목에 들어감). 초기순번 J는 TCP처럼 data chunk의 순번을 매길 때 쓰인다. 상대방도 initiation tag(Tz), 초기순번(K)와 함께 cookie C를 INIT-ACK chunk에 포함하여 응답한다.

여기서 Cookie 정보가 중요한데, Cookie는 다음 3번째 COOKIE-ECHO 패킷에 포함되어야 하며, 이를 통해 서버는 client의 인증을 수행한다. 이를 통해 TCP SYN-Flooding DoS(denial of service) attack 문제를 해결할 수 있다. 이처럼 SCTP의 four-way handshake는 DoS 공격에 대항하기 위해 Cookies 기법을 사용한다. 마지막으로 서버는 COOKIE-ACK를 전송함으로써 SCTP association 설정이 종료된다.

Association Termination

TCP와는 다르게 SCTP는 "half-closed"를 수행하지 않는다. 한 쪽이 association을 끊으면 반대편에서는 새로운 data의 전송을 중단해야 한다. 연결종료 요청이 수신 측에 보내지면 data는 queue에 대기되고 연결은 종료된다. 그림 2.6에 이러한 절차가 나타나있다.



<그림 2.7> SCTP 연결 종료

SCTP는 initiation tag를 통해 association를 구분할 수 있기 때문에 TCP처럼 TIME_WAIT 상태를 가질 필요가 없다. 모든 chunk들은 INIT chunk 및 INIT-ACK chunk에 포함되어 있는 initiation tag를 포함해야 한다. 연결이 종료된 후 다시 생성되는 연결에서의 tag는 다른 값을 가지기 때문이다. 이를 통해 SCTP는 TCP에 비해 구현성능을 강화시킨다.

SCTP State Transition Diagram

그림 2.8은 SCTP의 상태천이도(State transition diagram)를 보여준다.

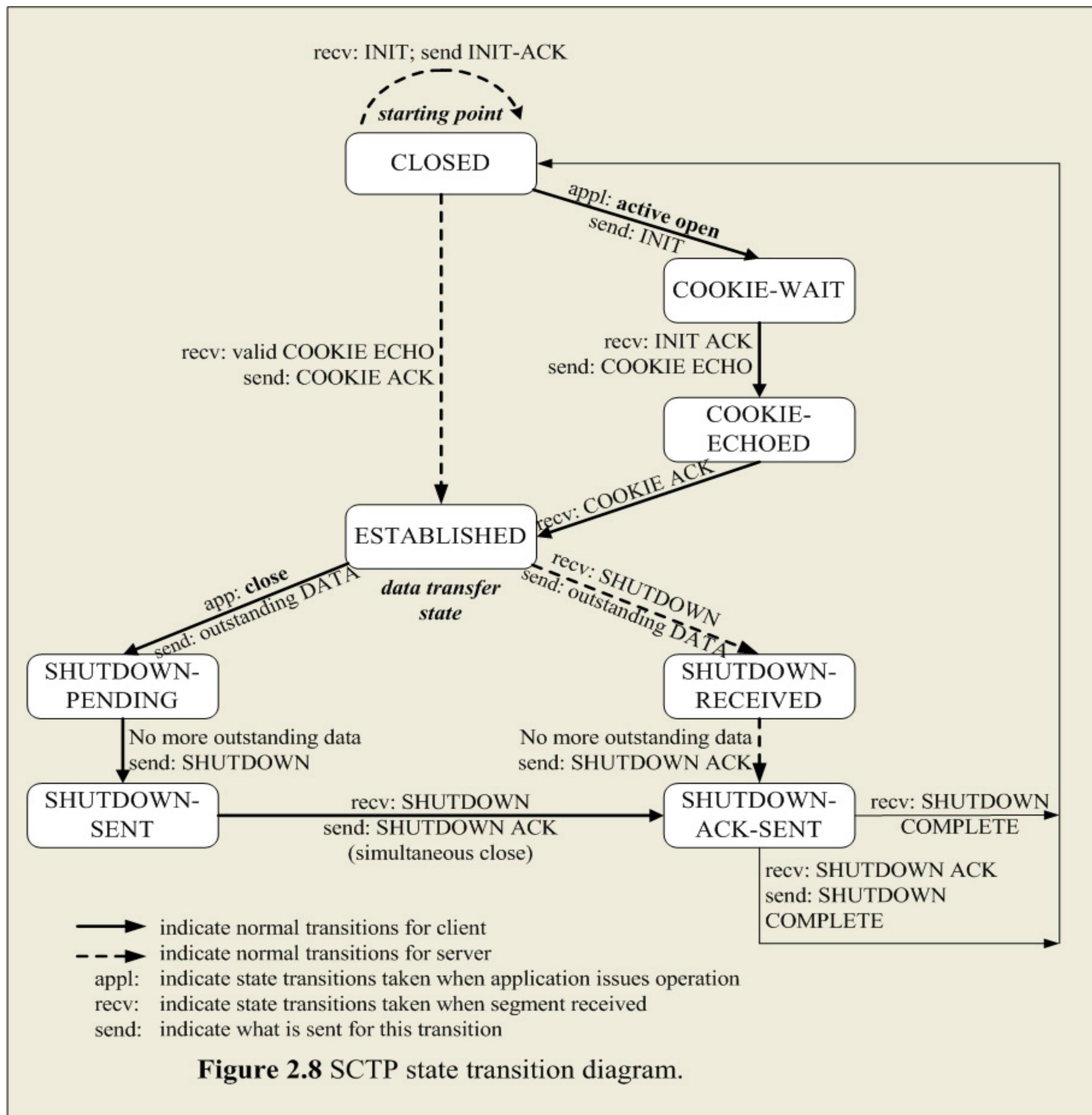


Figure 2.8 SCTP state transition diagram.

<그림 2.8> SCTP 상태천이도

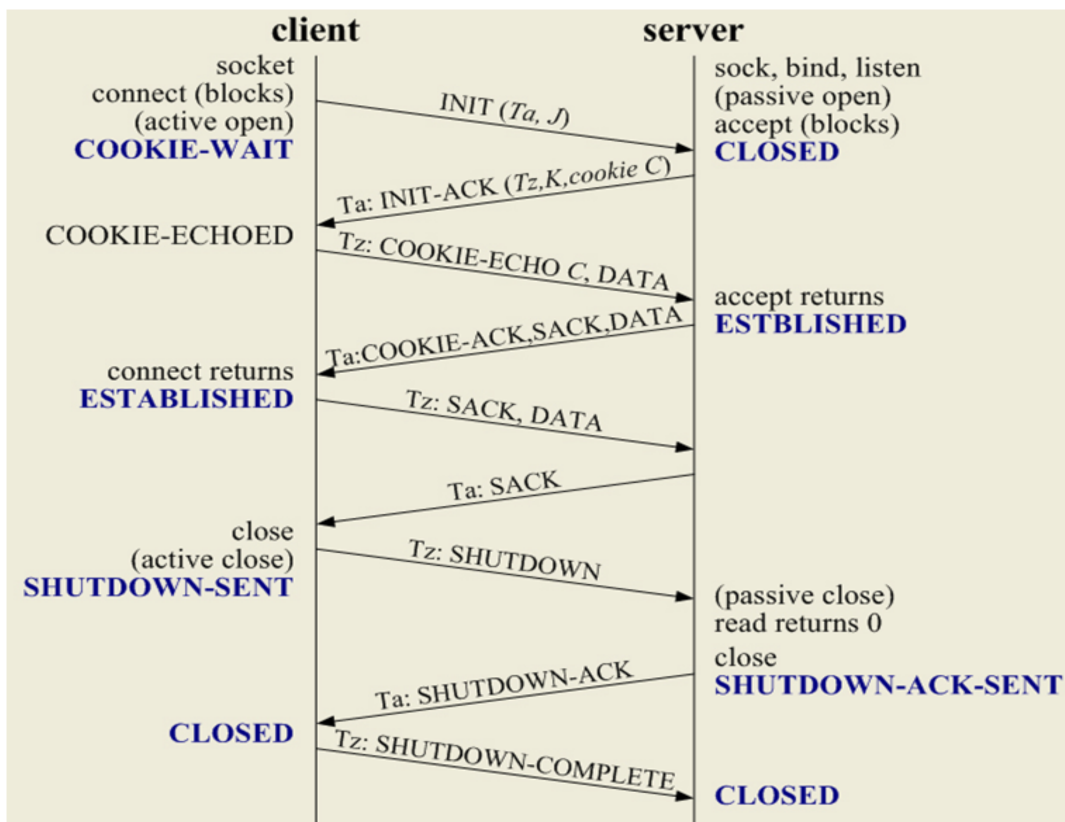
그림에서 상태의 천이는 현재 상태에서 받는 chunk 종류에 기반하여 이루어진다. 예를 들면, application이 CLOSED 상태에서 active open을 수행한다고 했을 때 SCTP는 INIT를 보내고 새로운 상태인 COOKIE-WAIT 상태가 된다. 다음에 SCTP가 INIT-ACK를 받으면 COOKIE-ECHO를 보내주고 새로운 상태 COOKIE-ECHOED 상태가 된다. SCTP가 COOKIE-ACK를 받으면 ESTABLISHED 상

태가 된다. 참고로, COOKIE-ECHO와 COOKIE-ACK chunk는 데이터 chunk를 piggybacking 할 수 있다.

ESTABLISHED 상태에서 나오는 두 화살표는 연결의 종료를 위함이다. 만약 application이 active close를 수행하면 남아있는(outstanding) DATA chunk를 마저 보내고 SHUTDOWN-PENDING 상태로 천이가 일어나며, SHUTDOWN chunk를 전송한 후에 SHUTDOWN-SENT 상태로 변경된다. 반대로 ESTABLISHED 상태에서 SHUTDOWN을 받았다면(passive close), SHUTDOWN-RECEIVED 상태로 전이가 일어나고 SHUTDOWN-ACK chunk로 응답하게 된다. 이후에 SHUTDOWN-ACK-SENT로 상태천이가 일어나고 (사실상, 이 시점에서 모든 상태정보는 release됨), 상대방으로부터 SHUTDOWN-COMPLETE chunk를 받게 되면 연결이 완전히 종료된다.

Watching the Packets

그림 2.9에서 SCTP 연결에서의 패킷 교환 예제를 보여준다.



<그림 2.9> SCTP 연결에서의 패킷 교환

그림에서 client는 COOKIE-ECHO에 첫 번째 data chunk를 piggyback하고 COOKIE-ACK에 server의 data와 응답을 받는다. 일반적으로 COOKIE-ECHO는 one-to-many 타입의 socket interface를 사용하여 application을 구현하는 경우에 data chunk를 포함할 수 있다. 반면에 one-to-one 타입의 socket 에서는 data chunk를 포함하지 않는다. 이에 대해서는 9장에서 다룬다.

SCTP의 기본 정보 단위는 'chunk'이다. 'chunk'는 chunk type, chunk flags, chunk length 등의 필드값으로 표현되며, 크게 DATA chunk와 다양한 control chunk들로 구분된다. 하나의 SCTP packet은 여러 개의 data/control chunk들을 포함할 수 있다.

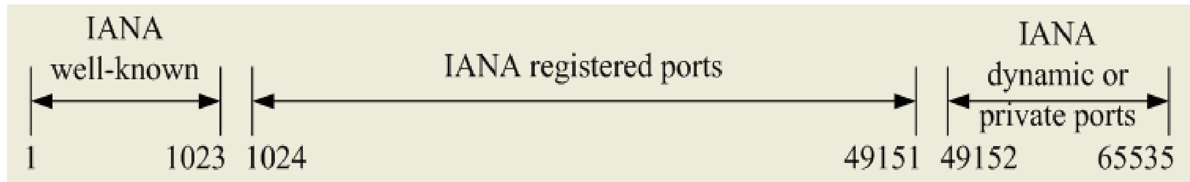
2.7 포트 번호

하나의 시스템에 다수의 process가 실행될 수 있으며, transport protocol(TCP, UDP, SCTP)들은 16-bit 정수의 port 번호를 사용하여 각 process를 구별한다. 대개 TCP, UDP, SCTP로 구현되는 서비스(응용)들은 '서버' 측에서 well-known port를 사용한다. 예를 들면, FTP 서버의 경우 TCP well-known port number 21번을 이용하고, TFTP(Trivial FTP) server의 경우 UDP port 69번을 사용한다. 반면에 'client'는 일반적으로 임시적인 ephemeral (dynamic or private) port를 사용한다. 일반적으로 client는 port는 client host에서 유일한 값이기만 하면 된다.

Internet Assigned Number Authority (IANA) 기구는 port 번호 할당을 관리한다. 할당 목록은 RFC 1700에 나와 있다. http://www.iana.org/에서 On-line database를 볼 수 있다. Port 번호는 다음 3가지 영역으로 구분된다.

1) Well-known port: 0 ~ 1023. 이 port번호들은 IANA에 의해서 할당 되고 관리되는 번호들이다. 가능하면 TCP, USP, SCTP에 같은 port 번호를 부여한다. 예를 들면, 지금은 모든 구현이 TCP를 사용하지만, TCP와 UDP 모두 Web server에 port 80을 부여한다.
2) Registered port: 1024 ~ 49151. IANA에 의해 관리되지는 않지만, 편의상 IANA에 의해 list에 등록되어 사용되는 port들이다. 가능하면 TCP와 UDP가 같은 port를 할당 받는다. 예를 들면, port 6000번에서 6063번 까지는 두 protocol 모두에게 X window server에 할당 되어 있지만, 최근에는 모든 구현에서 TCP에서만 사용된다.
3) Dynamic or Private port: 49152 ~ 65535. IANA에서는 이 port들에 대해 관여하지 않는다. 이 번호들은 주로 client 측의 port로 사용된다. (49152는 65536의 3/4에 해당한다)

그림 2.10 은 이러한 port 번호들의 할당을 보여준다.



<그림 2.10> 포트번호의 종류

Socket pair

TCP 연결에서 Socket pair는 연결 양쪽 endpoint를 정의하는 local IP 주소, local port, 원격지 IP 주소, 원격지 port이다. 이러한 socket pair는 network에서 모든 TCP 연결을 구분 지어준다. SCTP 에서는 local IP 주소의 집합, local port, 원격지 IP 주소의 집합, 원격지 port가 그 역할을 한다. TCP의 경우와 유사하나 multi-homed라는 점이 다르다.

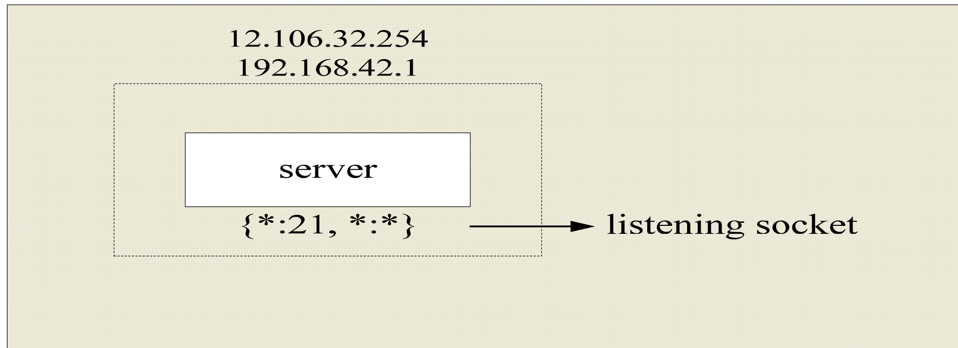
각 endpoint를 구분하는 IP 주소와 port 번호의 조합을 우리는 "socket"이라 부른다.

비록 UDP가 connectionless protocol이긴 하지만 socket의 개념을 UDP에 확장하여 적용할 수 있다. Socket 함수들(bind, connect, getpeername 등)을 설명할 때, 어떤 함수가 소켓에 어떤 값을 지정(대입)하게 되는지를 주목해야 한다. 예를 들면, TCP, UDP, SCTP에서 bind() 함수는 local IP 주소와 local port를 필요로 한다.

2.8 CONCURRENT 서버

앞서 언급한 iterative server에 대응되는 서버가 concurrent 서버이다. UDP와 달리 대개 TCP와 SCTP는 concurrent 서버 형태로 구현된다. Concurrent server에서는 server가 여러 client에 대하여 각각의 새로운 연결을 처리할 child process를 만드는데 만약 이 기간 동안에 child가 잘 알려진 port 번호를 계속해서 사용한다면 어떤 일이 발생하겠는가? 먼저 다음 예를 생각해 보자.

먼저 server는 freebsd라는 host에서 시작하고 12.106.32.254와 192.168.42.1같은 여러 IP 주소를 가지며 잘 알려진 port 번호(이 예제에서는 21)로 passive open을 한다. 그리고 나서 그림 2.11과 같이 client의 응답을 기다린다.

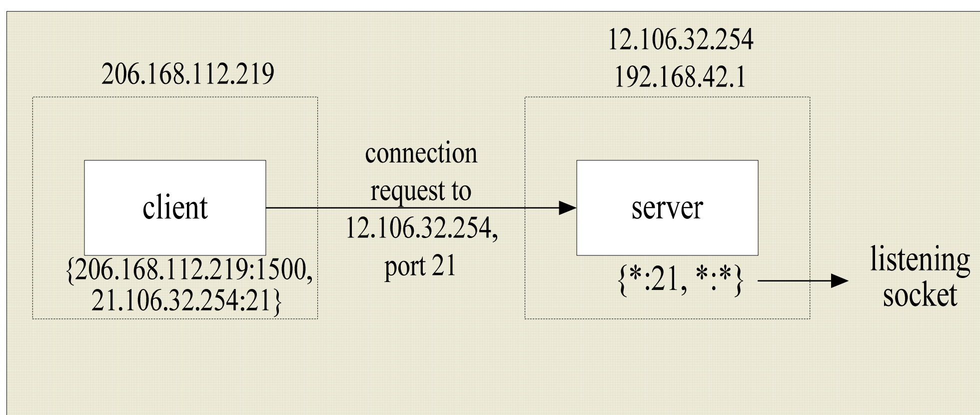


<그림 2.11> Concurrent Server 예제

{8:21, *:*}는 socket pair을 나타낸다. Server는 port 21의 어느 local interface(첫 번째 *)에 bind를 하고 client의 연결요청을 기다린다. 원격지 IP 주소와 port는 지정되지 않았기 때문에 *.*로 표기한다. 이것을 listening socket이라 한다. IP 주소와 port 번호를 구분하기 위해 : 부호를 사용한다 (HTTP 예제처럼).

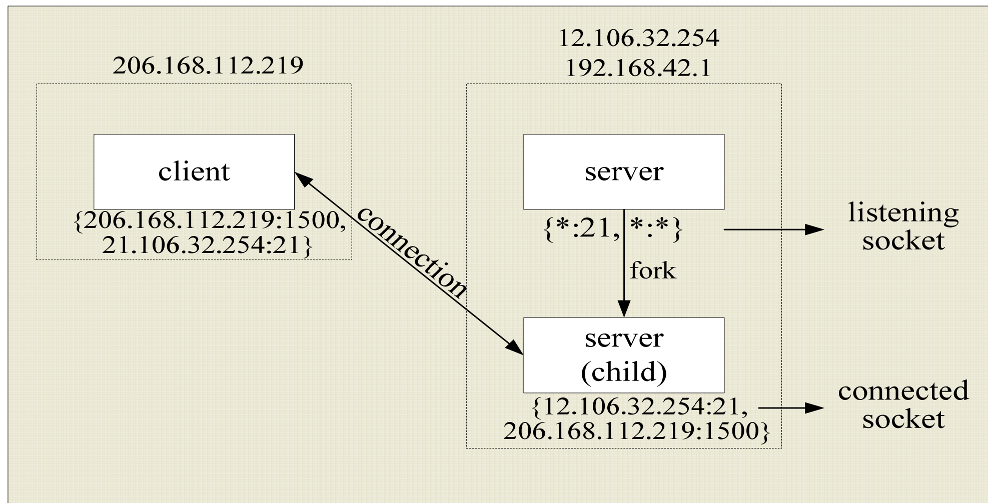
별표로 local IP 주소를 지정하면 이를 임의(wildcard) 문자라 부른다. Server가 동작하는 host가 multi-homed인 경우(예제처럼), server는 특정 지역의 접속만 허용하도록 할 수 있다. 이것은 server가 임의의 선택을 하는 것이다. Server는 다수의 주소를 지정하지 못한다. 임의의 local 주소가 '어떤(*)' 것을 선택하는 것이다. 1장의 예제에서는 bind()를 호출하기 전에 socket 주소 구조체의 IP 주소를 INADDR_ANY로 지정하여 임의 주소를 정하였다.

잠시 후, client는 IP 주소가 206.168.112.219인 host에서 시작하여 주소가 12.106.32.254인 server에 active open을 행한다. 여기서 dynamic port는 1500이라 가정한다. 이것은 그림 2.12에 잘 나타나 있다. Client의 아래에 socket pair가 보인다.



<그림 2.12> client의 접속

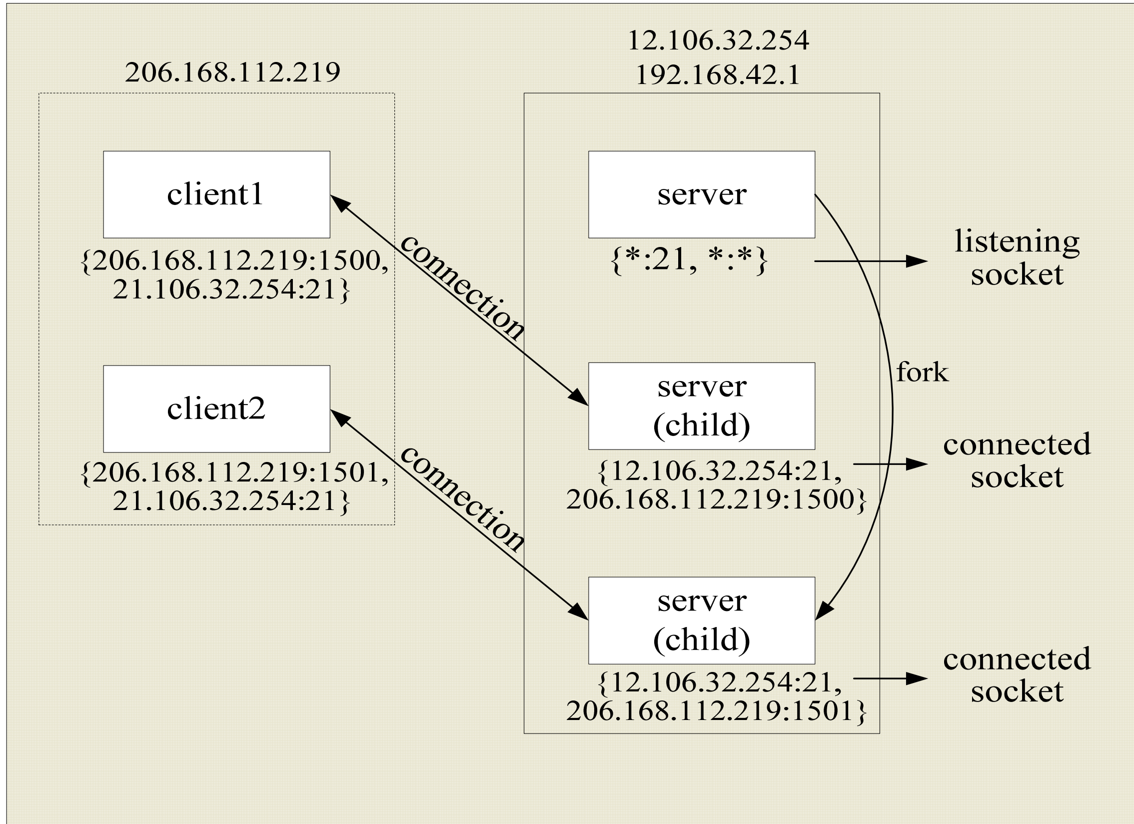
다음에 그림 2.13에서 보이듯, server가 client의 요청을 받고 허락하면 fork() 함수를 사용하여 자기 자신의 복제본(child process)을 만드는데 이 child process가 client를 관리한다. (fork 함수에 대해서는 4장에서 다룬다.)



<그림 2.13> child process의 생성

이쯤에서 우리는 server host의 listening socket과 connected socket을 구별하여야 한다. Connected socket은 listening socket과 같이 21번 port를 사용한다. Multi-homed의 경우 connected socket이 구체적인 local IP 주소(12.106.32.254)로 채워진다.

다음 단계로, 같은 client host에서 또 다른 client process가 같은 server에 연결을 요청한다고 가정한다. 이 때 새로운 client socket은 다른 dynamic port 번호를(1501) 할당한다. 이러한 시나리오가 그림 2.14에 나타나 있다. Server에서 두 연결은 구분이 된다. 첫 연결의 socket pair는 두 번째 연결의 socket pair와 다른데, 이는 client의 두 번째 연결에서 다른 dynamic port(1501)를 선택했기 때문이다.



<그림 2.14> 다른 client socket의 접속

이 예에서 서버에서는 도착한 TCP segment의 해당 endpoint를 결정하려면 socket pair에 있는 4개의 원소를 모두 보아야 한다. 그림 2.14에는 같은 local port(21)를 가지는 3개의 socket이 있다 (2개의 connected socket과 1개의 listening socket). 만약 {206.168.112.219, 1500}로부터 {12.106.32.254, 21}로 향하는 segment가 도착하면 첫 번째 child에게 전달된다. 반면에, {206.168.112.219, 1501}에서 {12.106.32.254, 21}로 향하는 segment가 도착하면 두 번째 child에게 전달된다. Port 21로 향하는 기타 다른 segment들은 listening socket으로 전달된다.

2.9 소켓의 송신버퍼

많은 network는 하드웨어에 의해 정해지는 MTU를 가지고 있다. 예를 들어 Ethernet MTU의 경우 1,500 byte이다. 두 host 사이에서 가장 작은 MTU를 경로(path) MTU라고 부른다. 오늘날에는 1,500byte인 Ethernet MTU가 경로 MTU인 경우가 많다. Internet에서 경로 배정은 비대칭적이므로 양방향의 경로 MTU가 같을 필요는 없다. 다시 말해 A에서 B로 가는 경로와 B에서 A로 가는 경로가 다를 수 있다는 말이다.

IP datagram을 보낼 때, 그 크기가 link의 MTU보다 크면 IPv4와 IPv6에서 분할(fragmentation)한다. 분할된 조각들은 최종 목적지에 도달 할 때까지 다시 합쳐지지 않는다. IPv4 host는 datagram을 생성 할 때 분할하고, IPv4 router는 datagram을 전송할 때 분할한다. 하지만 IPv6의 경우 datagram을 생성 할 때만 분할하고 IPv6 router는 datagram을 전송 할 때 분할하지 않는다.

만약 IPv4 header의 "don't fragment"(DF)가 설정되어 있으면 송신측 host나 어떤 router에 의해서도 분할되어서는 안 된다. DF가 설정되어있고 link의 MTU를 초과하는 datagram에 대해서는 ICMPv4 프로토콜에서 "destination unreachable, fragmentation needed but DF bit set"이라는 error message를 보낼 것이다. IPv6 router는 분할을 하지 않기 때문에 모든 datagram에 DF가 설정된 것과 같다. IPv6가 link의 MTU를 초과하는 datagram을 받으면 ICMPv6가 "packet too big"이라는 error message를 보낸다.

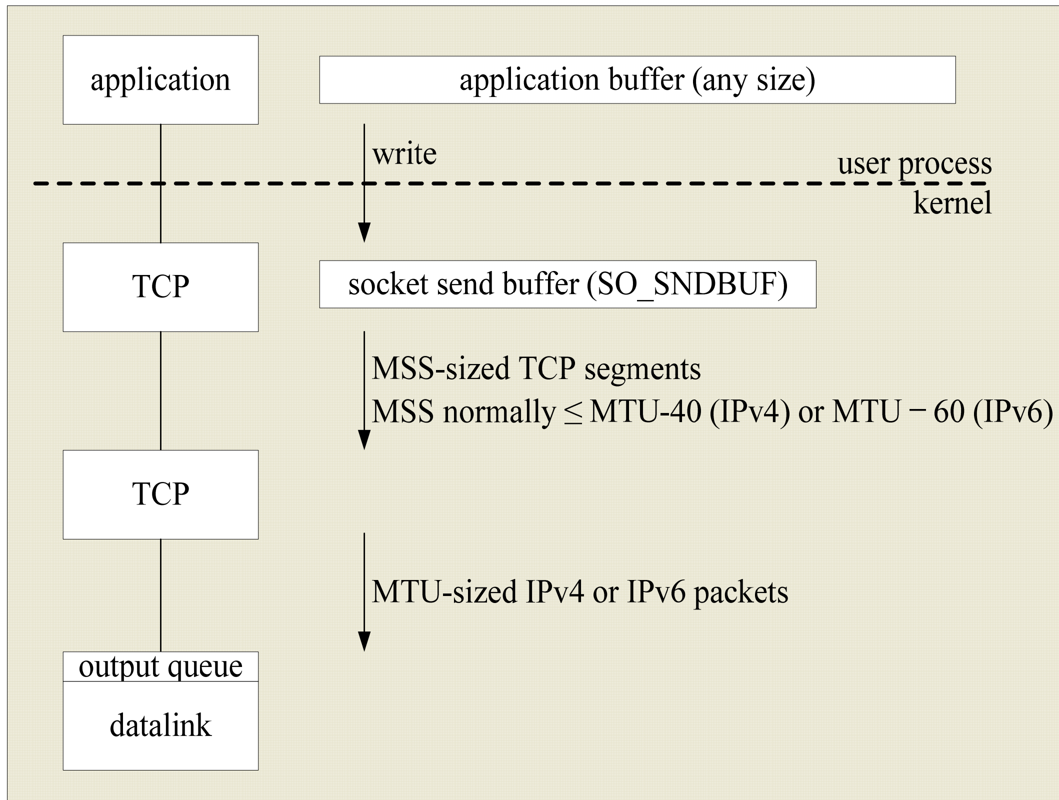
TCP는 상대방에 보낼 최대 data 크기를 알려주는 MSS(Maximum Segment Size)가 있다. MSS의 목적은 상대방에게 내가 보낼 데이터 크기의 실제 값을 알려주고 되도록 패킷 분할을 피하는 것이다. MSS는 link MTU에서 IP header와 TCP header 크기를 뺀 값으로 한다. 1500 byte의 MTU를 사용하는 Ethernet에서, IPv4 MSS는 1,460 byte이고 IPv6 MSS는 1,440 byte이다 (TCP header 20 byte, IPv4 header 20 byte, IPv6 header는 40 byte이기 때문에).

TCP Output

그림 2.15에서는 application이 TCP socket에 data를 쓸 때 일어나는 일을 보여주고 있다.

모든 TCP socket에는 송신 buffer가 있고 SO_SNDBUF socket option을 사용해 그 크기를 바꿀 수 있다(7장). Application이 write()를 호출 했을 때, kernel이 application buffer의 모든 data를 socket 송신 buffer로 복사한다. 만약 socket buffer가 모든 application buffer의 data를 담을 수 없으면 (application buffer가 socket 송신 buffer보다 크거나, socket 송신 buffer에 이미 다른 data가 들어와 있는 경우) process는 수면상태(sleep)가 된다. 이는 socket의 기본 동작이다.

Application buffer의 마지막 byte가 socket 송신 buffer로 복사 될 때까지 write() 함수는 반환되지 않는다. 즉, TCP socket에 write()하는 것에서 성공적인 복귀를 하면 application buffer를 재사용 할 수 있게 된다. 하지만 이것만으로 상대방 TCP가 data를 받았다거나 상대방 application이 data를 받았다고 장담할 수는 없다. 수신자의 성공적인 수신여부는 TCP 프로토콜의 오류제어 기법에서 알 수 있고, 이는 소켓 프로그래밍과는 독립적이다.



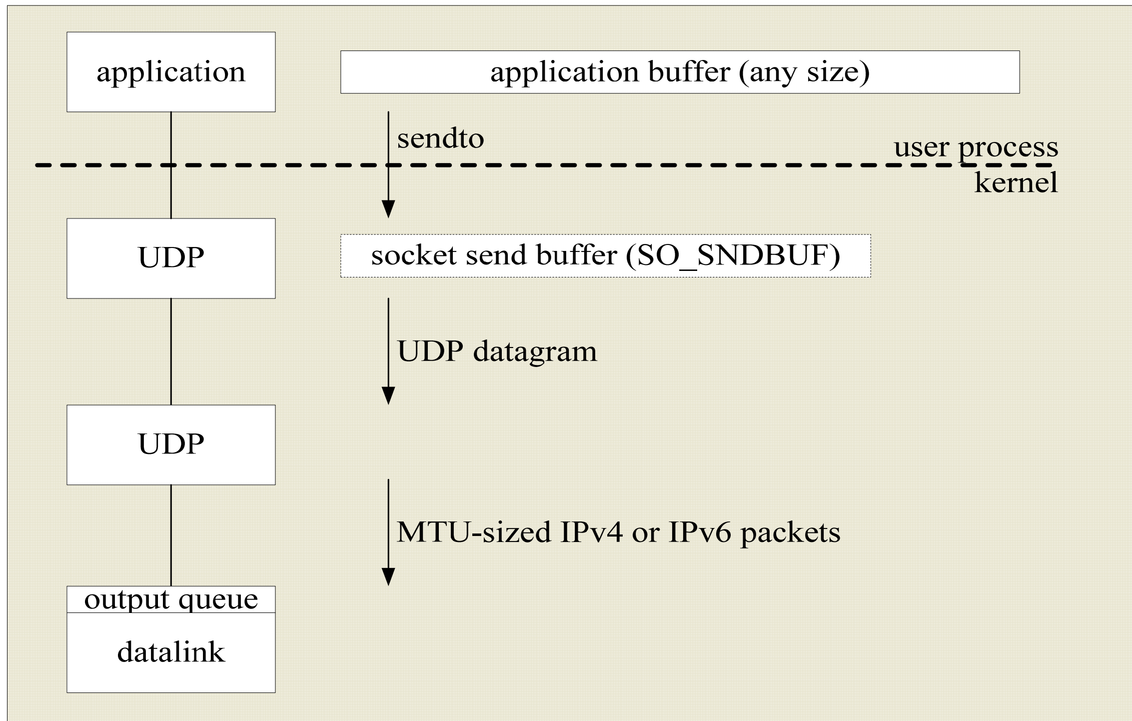
<그림 2.15> TCP 송신과정과 소켓

TCP는 data 전송(transmission) 메커니즘에 의거하여 socket 송신 buffer에 있는 data를 상대방 TCP로 보낼 것이다. 상대방의 TCP는 data 수신에 대한 확인(acknowledgment)을 해줘야 하며 TCP는 이 ACK를 수신해야 TCP socket 송신 buffer를 비울 수 있다. TCP는 상대방의 도착 승인이 있을 때까지 data의 복사본을 가지고 있어야 한다.

TCP는 data를 MSS 만큼씩 segment로 구성하여 상대방에게 전송하는데, 이 때 MSS는 상대방이 초기화 가정에서 알려준 값이거나 default로 536 byte를 사용할 수 있다. IP는 자신의 header를 앞쪽에 덧붙이고 routing table에서 목적지의 IP 주소 찾아(routing table의 항목에서 interface밖으로 나가는 것을 찾는다) 적당한 link로 보낸다. IP는 datagram을 datalink로 보내기 전에 분할할 수 있다. 각 datalink는 출력 queue를 가지고 이 queue가 가득 차면 packet을 잃게 되고 오류를 protocol stack의 위쪽으로(datalink에서 IP로, IP에서 TCP로) 보낸다. TCP는 이 error를 기억하였다가 차후에 그 segment를 다시 보낸다. 이런 일시적 상황은 application에는 보고 되지는 않는다.

UDP Output

그림 2.16은 application이 UDP socket에 data를 쓸 때 무슨 일이 일어나는지 보여주고 있다.



<그림 2.16> UDP 송신과정과 소켓

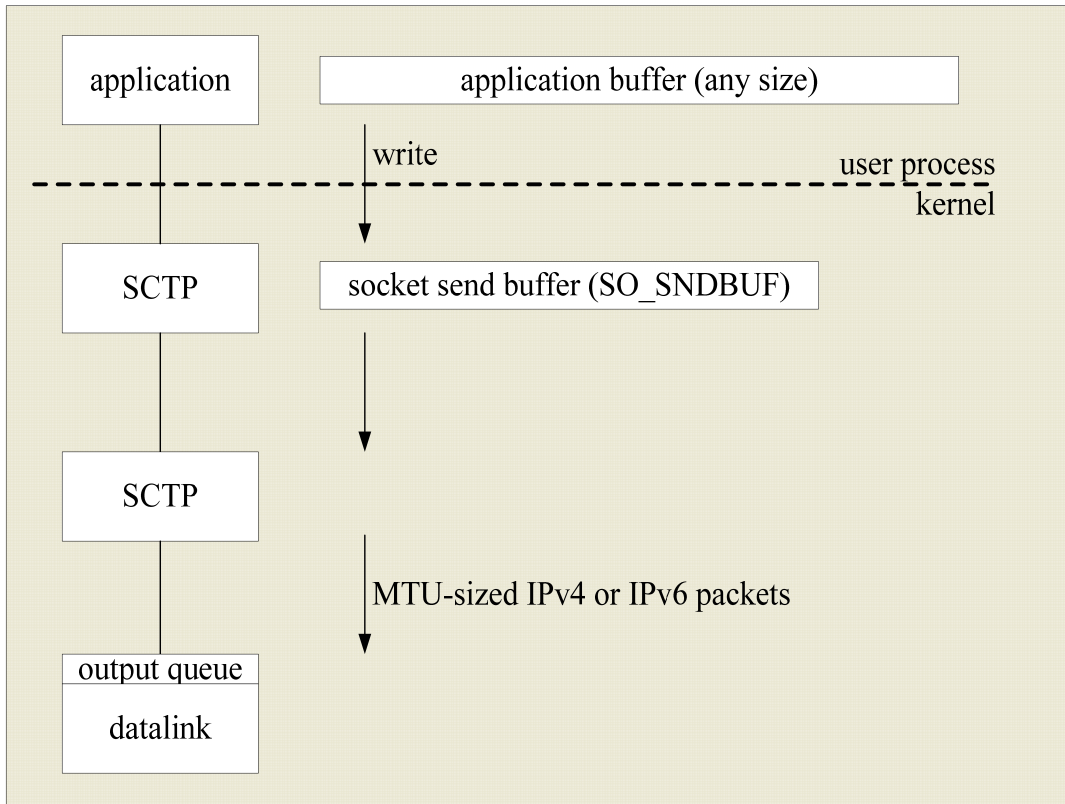
UDP에는 socket 송신 buffer가 실제로 존재하지 않으므로 점선 사각형으로 그렸다. UDP socket은 송신 buffer 크기를 가지고 있으나 이는 단순히 socket에 쓰일 UDP datagram의 최대 크기일 뿐이다. 만약 application이 socket 송신 buffer크기 보다 더 큰 datagram을 쓴다면 EMSGSIZE를 반환 받게 된다. UDP는 신뢰성이 없으므로 application의 data의 복사본을 가지고 있을 필요도 없고 실제로 송신 buffer도 필요 없다.

UDP는 간단히 datagram의 앞에 8 byte의 header를 붙여 IP에 전달한다. IPv4나 IPv6는 자신의 header를 덧붙이고 routing 함수를 수행하여 내보낼 interface를 정한 뒤 datagram을 datalink 출력 queue에 넣거나(MTU보다 작은 경우) datagram을 분할해서 각 조각을 datalink 출력 queue에 넣는다. TCP에서는 application data를 MSS 단위로 나누지만 UDP는 그럴 필요가 없다.

UDP socket에 write가 성공적으로 끝나면 datagram이나 datagram이 가진 모든 조각이 datalink 출력 queue에 들어갔음을 알 수가 있다. 만약 queue에 datagram이나 조각들을 받을 공간이 부족하다면 ENOBUFS를 application에 반환한다.

SCTP Output

그림 2.17은 application이 SCTP socket에 data를 쓸 때 무슨 일이 일어나는지 보여준다.



<그림 2.17> SCTP 송신과정과 소켓

SCTP는 TCP처럼 신뢰성이 있는 protocol이므로 송신 buffer를 가진다. TCP처럼 application이 SO_SNDBUF를 사용하여 buffer 크기를 조절하는 것도 가능하다(7장). Application이 write() 함수를 호출할 때, kernel은 application buffer의 data를 모두 복사하여 socket 송신 buffer에 넣는다. Application buffer의 마지막 byte가 socket buffer에 복사될 때까지 write에서 빠져나올 수 없다.

SCTP는 SCTP data 전송법칙(transmission)에 의거하여 socket 송신 buffer의 data를 상대방 SCTP로 보낸다. 송신측 SCTP는 socket buffer의 data를 지우기 전에 상대방 SCTP로부터 SACK를 받아야 한다.

3. 기본적인 SOCKETS

이 장에서는 socket API에 대하여 소개한다. 이 책의 거의 모든 예제에서 볼 수 있는 socket address 구조체부터 시작한다. 이 구조체는 두 가지로 분류된다: process에서 kernel로 가는 구조체 그리고 kernel에서 process로 가는 구조체. kernel에서 process로 가는 구조체의 경우 value-result argument가 그 예이다. 그리고 책을 통해서 이 argument에 대한 다른 예제를 다룰 것이다.

이 책에서는 Protocol에 무관한 방식으로 socket address 구조체에서 작동하는 sock_로 시작하는 함수들을 개발할 것이다. 이를 이용해서 책의 code를 protocol에 무관하게 사용할 것이다.

3.1 SOCKET ADDRESS 구조체

대부분 socket 함수는 argument로 socket address 구조체 포인터가 필요하다. 지원되는 protocol suite는 각각 자신의 socket address 구조체를 정의한다. 이 구조체의 이름은 sockaddr_로 시작해서 각각의 protocol suite의 특수한 접미사로 끝난다.

IPv4 Socket Address 구조체

일반적으로 "Internet socket address 구조체"라 불리는 IPv4 socket address 구조체는 <netinet/in.h> header를 포함함으로써 정의되고 socketaddr_in으로 불린다.

```
struct in_addr {
in_addr_t    s_addr;        /* 32-bit IPv4 address */
                        /* network byte ordered */
};
struct sockaddr_in {
  uint8_t        sin_len;    /* length of structure */
  sa_family_t     sin_family;  /* AF_INET */
  in_port_t      sin_port;   /* 16-bit TCP or UDP port number */
                        /* network byte ordered */
  struct in_addr   sin_addr;   /* 32-bit IPv4 address */
                        /* network byte ordered */
  char        sin_zero[8];  /* unused */
```

이 예제에서 사용하는 일반적인 socket address 구조체에 대해서 여러 가지 주의할 점이 있다.

- sin_len, length member는 첨부된 OSI protocol을 지원할 때, 4.3BSD-Reno와 함께 첨부된다. sin_len 배포 전에는, 첫번째 member는 unsigned short인 sin_family였다. 모든 vender가 socket address 구조체를 지원하기 위해 length field를 지원하지는 않고, POSIX 표준은 이 member가 필요가 없다. 우리가 사용하는 uint8_t가 전형적인 datatype이다. 그리고 POSIX 산하 시스템들은 이 형식의 datatype을 제공한다.
- 가변 길이 socket address 구조체를 다루기 위한 간단하게 만든 length field가 존재한다.
- 비록 length field가 존재할지라도, 만약 routing socket을 다루지 않는다면 절대 조사하지도 설정하지도 않는다. 다양한 protocol군(예를 들면 routing table code)에서 socket address 구조체를 다루는 routine에 의해 kernel에서 사용된다.
- process에t서 kernel로 socket address 구조체를 전달하는 4가지 함수(bind, connect, sendto, sendmsg)는 모두 Berkeley에서 구현된 sockargs 함수에서 사용이 된다. 이 함수는 process에서부터 socket address 구조체를 복사하고, 이 네 가지 함수에서 인자로 사용되는 구조체의 크기로 함수의 sin_len member로 설정된다. Kernel에서 process로 socket address 구조체를 전달하는 5개의 socket 함수(accept, recvfrom, recvmsg, getpeername, getsockname)는 모두 process에서 return되기 전에 sin_len member를 설정한다.
- 유감스럽게도 일반적으로 socket address 구조체를 위한 length field를 정의한 구현인지 아닌지 결정하기 위한 compile-time test는 간단하지가 않다. Code에서 HAVE_SOCKADDR)SA)LEN 상수를 테스트한다. 그러나 이 상수를 정의한지 안 한지는 컴파일이 성공하느냐 못하느냐 살펴 보고 이 선택적은 구조체 member를 사용하는 간단한 테스트 프로그램을 컴파일 함으로써 알 수 있다. 만약 socket address 구조체가 length field를 가졌다면 IPv6 구현은 SIN6_LEN을 정의할 것을 요구하는 예제를 볼 것이다. 몇몇의 IPv4구현은 compile-time option(예를 들면 _SOCKADDR_LEN)으로 socket address 구조체의 length field 적용을 제공한다. 이 특징은 오래된 프로그램을 위한 호환성을 제공한다.
- POSIX 표준은 구조체에서 단지 3개의 member만 필요로 한다: sin_family, sin_addr, sin_port. POSIX 호환 구현이 추가적인 구조체 member를 정의하는 것을 허용하고, 이는 Internet socket address 구조체를 위한 일반적인 것이다. 거의 모든 구현은 sin_zero member을 추가한다. 그래서 모든 socket address 구조체는 크기가 적어도 16byte이다.
- s_addr, sin_family, sin_port member를 위한 POSIX datatype를 보여준다. in_addr_t datatype은

최소한 32bit의 unsigned integer type임에 틀림없고, sa_family_t는 unsigned integer type일 수 있다. 만약 구현이 length field를 지원한다면, 후자는 일반적으로 8 bit unsigned integer, 혹은 16 bit unsigned tnteger이다.

➢ 다음은 몇몇 다른 POSIX datatype 사이의 세가지 POSIX 정의된 datatype의 표이다.

Datatype	Description	Header
int8_t uint8_t int16_t uint16_t int32_t uint32_t	Signed 8-bit integer Unsigned 8-bit integer Signed 16-bit integer Unsigned 16-bit integer Signed 32-bit integer Unsigned 32-bit integer	<sys/types.h> <sys/types.h> <sys/types.h> <sys/types.h> <sys/types.h> <sys/types.h>
sa_family_t socklen_t	Address family of socket address structure Length of socket address structure, normally uint32_t	<sys/socket.h> <sys/socket.h>
in_addr_t in_port_t	IPv4 address, normally uint32_t TCP or UDP port, normally uint16_t	<netinet/in.h> <netinet/in.h>

➢ 모두 unsigned인 u_char, u_short, u_int, u_long datatype들도 언급할 것이다. POSIX 표준은 진부한 주석으로 이를 정의하고 있다. 거꾸로 호환성을 위해 그 datatype들을 제공한다.

➢ IPv4 주소와 TCP 혹은 UDP port 번호 둘 다 network byte 순서로 구조체에 항상 저장된다. 이 member들이 사용할 때 이들을 인식하고 있음에 틀림없다.

➢ 32-bit IPv4 address는 두 가지 다른 방법으로 access될 수 있다. 예를들면, 만약 serv가 Internet socket address 구조체로 정의되었다면, serv.sin_addr은 in_addr 구조체로써 32-bit IPv4 address를 참조한다, 반면에 serv.sin_addr.s_addr은 in_addr_t(전형적으로 unsigned 32-bit integer)로 같은 32-bit IPv4 address를 참조한다. 특히 함수의 인자로 사용될 때, IPv4 address를 확실히 참조하고 있다고 확신한다. 왜냐하면 컴파일러는 종종 integer로부터 다른 구조체를 통과시키기 때문이다.

➢ in_addr_t가 아닌 sin_addr member가 구조체인 이유는 고전적이다. 다양한 구조체의 union으로 in_addr 구조체가 정의된 초창기 배포판(4.2 BSD)은 32-bit IPv4 address를 내포한 16-bit

값과 4byte 각각의 access를 허용한다. 이는 address의 적합한 byte를 가지고 온 class A, B, C에서 사용된다. 그러나 subnetting의 출현과 classless addressing으로 다양한 address class의 사라짐으로 인해 union의 필요성이 사라졌다. 오늘날 대부분의 시스템은 union을 가지지 않고 단지 하나의 in_addr_t member로 가진 구조체로써 in_addr을 정의한다.

- sin_zero member는 사용되지 않는다. 그러나 우리는 이 구조체 중의 하나를 채울 때 0으로 이를 설정한다. 관례에 의해 sin_zero member뿐 아니라 값을 채우기 전 모든 구조체를 0으로 설정한다.
- 비록 non-wildcard IPv4 address를 bind할 때, 대부분 사용하는 구조체는 이 member를 0으로 설정할 필요가 없지만, 이 member는 틀림없이 0이다.
- Socket address 구조체는 단지 주어진 host에서만 사용된다: 구조체 그 자체는 다른 host사이에서 통신하지 않지만, 특정한 field(예를 들면, IP address와 port)는 통신에 사용된다.

일반적인(generic) Socket Address 구조체

Socket address 구조체는 항상 어떤 socket 함수에서 인자로 전달될 때 reference에 의해 전달된다. 그러나 인자로써 이 포인터중의 하나를 가지는 어떤 socket 함수는 지원되는 protocol군의 어떤 함수로부터 socket address 구조체로 다루어져야 한다.

전달된 포인터의 type을 어떻게 선언할지 문제가 발생한다. ANSI C에서, 간단한 해결책은 간단하다: void*은 일반적인 pointer type이다. 그러나 socket 함수는 ANSI C와 1982년에 채택된 다음에서 주어진 <sys/socket.h> 헤더에서 일반적인 socket address 구조체를 정의하는 해결책보다 먼저 나왔다.

```
struct sockaddr {
  uint8_t       sa_len;
  sa_family      sa_family;      /* address family: AF_xxx value */
  char          sa_date[14];  /* protocol-specific address    */
};
```

socket 함수는 ANSI C bind 함수를 위한 함수 prototype에서 보여진 것처럼, 일반적인 socket address 구조체를 포인터로 가지도록 정의된다:

```
int bind(int, struct sockaddr *, socklen_t);
```

이는 이 함수들을 호출하는 것은 일반적인 socket address 구조체에서 포인터가 되는 특정 protocol socket address 구조체에서 포인터로 캐스팅해야 한다. 예를 들면,

```
struct sockaddr_in    serv;       /* IPv4 socket address structure */
/*  fill in serv{} */
bind(sockfd, (struct sockaddr *) &serv, sizeof(serv));
```

만약 "(struct sockaddr *) 캐스팅을 빠뜨린다면, C 컴파일러는 시스템의 헤더가 bind 함수를 위한 ANSI C prototype을 가질 것을 추측하고, "warning: passing arg2 of 'bind' from incompatible pointer type,"이라는 warning을 생성한다.

application programmer의 관점에서, 단지 이 일반적인 socket address 구조체의 사용은 특정 protocol 구조체에서 포인터를 캐스팅하는 것이다.

1장에 있는 lnp.h 헤더를 다시 보면, 이 포인터 캐스팅하는 것은 단지 짧은 코드로 줄이기 위해서 "struct sockaddr"은 SA로 정의한다.

kernel의 관점에서, 인자로 일반적인 socket address 구조체를 포인터로 사용하는 또 다른 이유는 kernel이 호출자의, struct sockaddr *로 캐스팅한 포인터를 가져야 하기 때문이다. 그리고 나서 구조체의 type을 결정하기 위해 sa_family의 값을 살펴본다. 그러나 application programmer의 관점에서 이는 더 간단하다. 만약 포인터 type이 void* 이라면, 명백히 캐스팅할 필요성이 사라진다.

IPv6 Socket Address 구조체

IPv6 socket address 구조체는 <netinet/in.h> 헤더에 포함되어 정의되고 다음에 나와있다.

```
struct in6_addr {
  uint8_t  s6_addr[16];               /* 128-bit IPv6 address */
                                 /* network byte ordered */
};
#define SIN6_LEN     /* required for compile-time tests */

struct sockaddr_in6 {
  uint8_t          sin6_len;          /* length of this struct  (28)  */
  sa_family_t       sin6_family;        /* AF_INET6 */
  in_port_t         sin6_port;          /* transport layer port# */
                                 /* network byte ordered */
  uint32_t          sin6_flowinfo;      /* flow information, undefined */
  struct in6_addr   sin6_addr;          /* IPv6 address */
                                 /* network byte ordered */
  uint32_t         sin6_scope)id;      /* set of interfaces for a scope */
};
```

- 만약 시스템이 socket address 구조체를 위해 length member를 지원하면, SIN6_LEN 상수는 정의되어야 한다.
- IPv6 family는 AF_INET6이다, 반면에 IPv4 family는 AF_INET이다.
- 이 구조체에서 member는 순차적이다. 그래서 만약 sockaddr_in6 구조체가 64-bit 정렬되었다면, 128-bit sin6_addr member이다. 몇몇 64-bit 프로세서 상에서 64-bit 값의 data는 64-bit 범위내에서 저장되어 최적화된다.
- sin6_flowinfo member는 두 분야로 나뉜다.
  - 내림차순 20bit는 flow label이다.
  - 오름차순 12bit는 예약되어있다.
- sin6_scope_id는 정해진 주소가 의미 있는 영역인지 확인한다. 일반적으로 대부분 link-local 주소를 위한 interface index이다.

새로운 일반적인 Socket Address 구조체

새로운 일반적인 socket address 구조체는 IPv6 socket API중의 일부로 정의 되고, struct sockaddr에 존재하는 극복한다. struct sockaddr과 달리, 새로운 struct sockaddr_storage는 시스템에서 지원되는 socket address 형태를 가지기에 충분히 크다. sockaddr_storage 구조체는 다음 코드에서 볼 수 있고, <netinet/in.h>헤더에 포함되어 정의되어 있다.

```
struct sockaddr_storage {
  uint8_t      ss_len;          /* length of this struct */
  sa_family_t   ss_family;        /* address family: AF_xxx value */
  /* implementation-dependent elements to provide:
   * a) alignment sufficient to fulfill the alignment requirements of
   * all socket address types that the system supports.
   * b) enough storage to hold any type of socket address that the
   * system supports.
   */
```

sockaddr_storage type는 struct sockaddr과는 다른 두 가지 방법으로 일반적인 socket address 구조체를 제공한다.

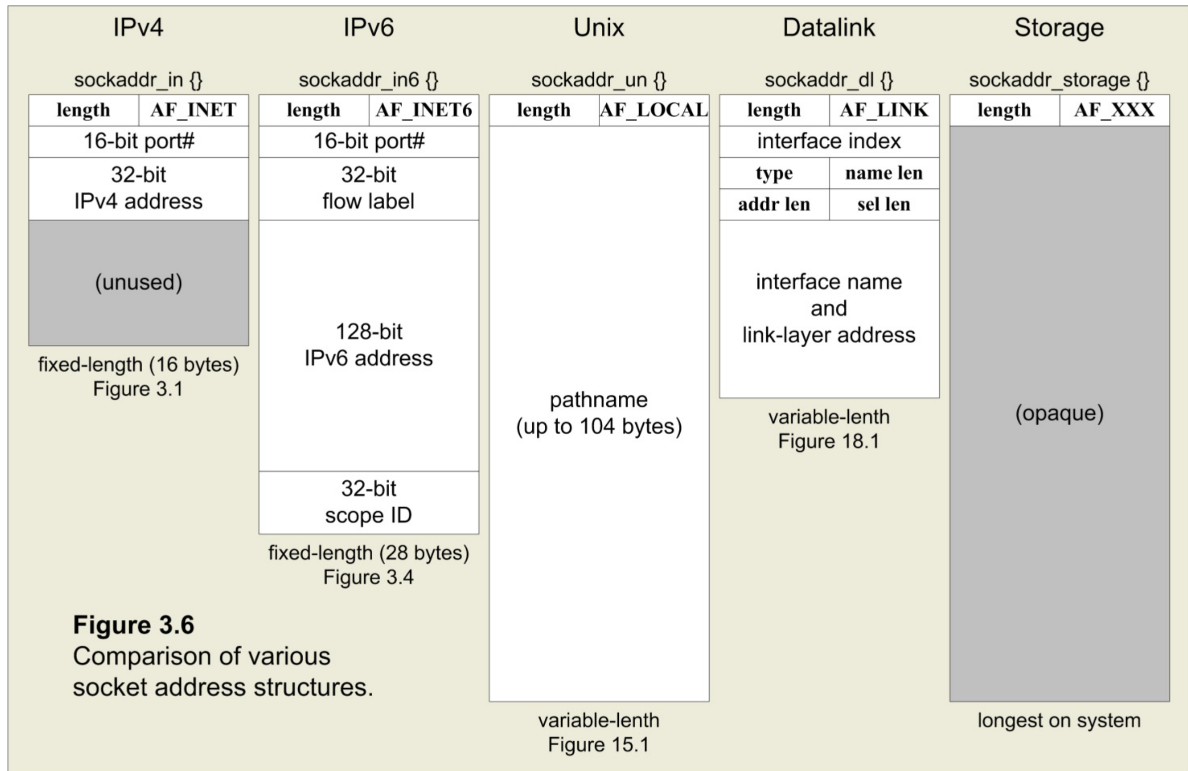
a) 만약 시스템에서 제공하는 어떤 socket address 구조체가 정렬 요구를 가진다면, sockaddr_storage는 아주 정밀한 정렬요구를 제공한다.

b) sockaddr_storage는 시스템에서 제공하는 어떠한 socket address 구조체를 포함하기에 충분히 크다.

sockaddr_storage 구조체의 field는 ss_family와 ss_len(만약 존재한다면)를 제외하고, 사용자에게 분명하지 않다. sockaddr_storage는 다른 field를 access하기 위해서 ss_family에서 주어진 주소를 위한 적합한 socket address 구조체로 복사되거나 casting되어야 한다.

Socket Address 구조체의 비교

다음 그림은 이 책에서 다룰 다섯 가지 socket address 구조체의 비교를 보여준다: IPv4, IPv6, Unix domain, datalink, storage. 이 그림에서, socket address 구조체는 한 byte length field를 포함한다. 그리고 family field는 1 byte를 차지한다. 그리고 최소한의 bit의 수를 가진 몇몇 field는 정확히 bit의 수이다.



<그림 3.1> 다양한 socket address 구조체

socket address 구조체 중에 두 가지는 길이가 고정되어있다, 반면에 Unix domain 구조체와 datalink 구조체는 가변 길이이다. 가변 길이 구조체를 다루기 위해서는, 언제라도 socket 함수의 하나에 인자로 socket address 구조체에 포인터를 넘기고, 다른 인자로 길이를 넘기기도 한다. 각각의 구조체의 바로 밑에 고정 길이 구조체의 byte단위 길이(4.4BSD 구현)가 표시되어 있다.

Sockaddr_un 구조체 그 자체는 가변 길이가 아니다. 그러나 정보(구조체 내의 경로명)의 길이는 가변적이다. 이 구조체에 포인터를 전달할 때, socket address 구조체에서 이 길이(구현에 의해 지원된다면)와 kernel로부터 길이를 어떻게 처리할지 주의해야 한다.

구조체 이름은 sockaddr_in{}처럼 항상 굵은 글씨로 나타낸다.

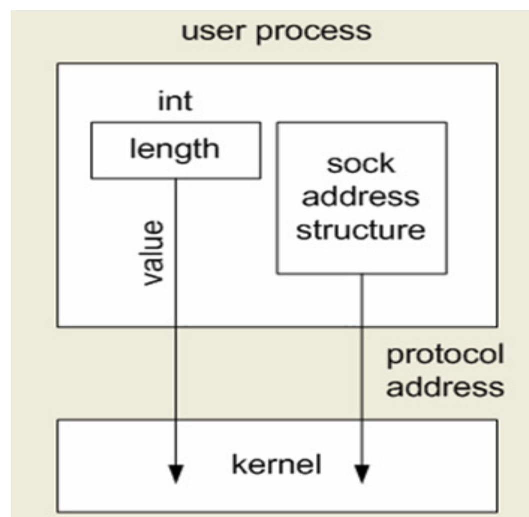
3.2 VALUE-RESULT ARGUMENTS

Socket address 구조체가 어떤 socket 함수로 전달될 때, 항상 reference에 의해 전달된다고 언급했다. 즉, 구조체 포인터로 전달이 된다. 구조체의 길이 역시 인자로써 전달된다. 그러나 구조체 길이가 전달되는 방향에 의존한다: process에서 kernel, 혹은 그 반대의 경우이다.

1. 세가지 함수, bind, connect, sendto,는 process로부터 kernel로 socket address 구조체를 전달한다. 이 세가지 함수에서 하나의 인자는 socket address 구조체 포인터이다. 그리고 다른 인자는 구조체의 정수형 크기이다.

```
sturct sockaddr_in  serv;
/* fill in serv{} */
connect (sockfd, (SA *) &serv, sizeof(serv));
```

kernel은 포인터와 포인터가 가리키는 것의 크기를 전달받기 때문에, process로부터 kernel로 복사되는 data의 정확한 크기를 안다. 다음 그림은 이 시나리오를 보여준다.



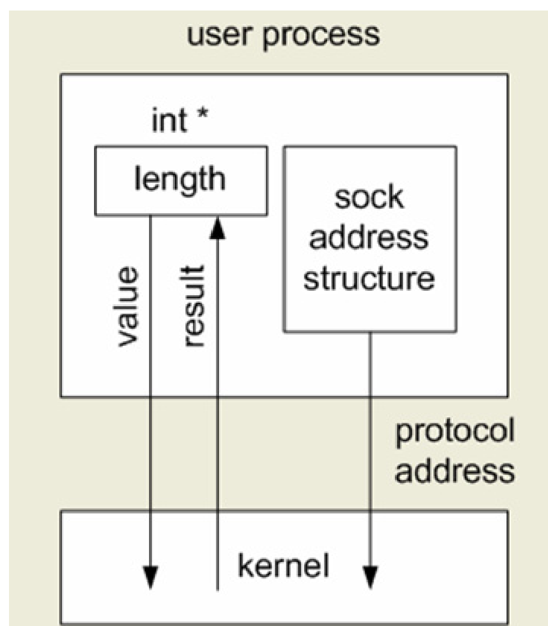
<그림 3.2> 프로세스에서 커널로의 socket address 구조체 전달

socket address 구조체의 크기를 위한 datatype이 실제로 int가 아닌 socklen_t라는 것을 다음 chapter에서 볼 것이다. 그러나 POSIX 표준은 socklen_t가 uint32_t로 정의 될 것을 추천한다.

2. 4가지 함수, accept, recvfrom, getsockname, getpeername은 이전 시나리오와는 반대로 kernel에서 process로 socket address 구조체를 전달한다. 이 네 가지 함수에서 인자 중에 두 개는 아래와 같이 구조체의 크기를 포함하는 정수형 포인터와 socket address 구조체 포인터이다.

```
struct sockaddr_un  cli;    /* Unix domain */
socklen_t    len;
len = sizeof(cli);        /* len is a value */
getpeername(unixfd, (SA *) &cli, &len);
/* len may have changed */
```

정수형에서 정수형 포인터가 되면서 크기가 변하기 때문에, 크기는 함수에서 불려지고(kernel이 구조체의 크기를 알기 때문에 kernel은 구조체의 끝에 크기를 쓰지 않는다.)크기는 함수에서 리턴된다 (실제로 구조체에서 저장된 지나간 kernel 정보의 양을 process로 보낸다). 이 인자의 형태를 value-result 인자로 불린다. 다음 그림은 이 시나리오를 보여준다.



<그림 3.3> 커널에서 프로세스로의 socket address 구조체 전달

process와 kernel 사이에 전달되는 socket address 구조체에 대하여 이야기 하고 있었다. 4.4 BSD처럼 구현하기 위해서, 모든 socket 함수는 kernel 안에서 시스템 콜을 한다. 그러나 몇몇에서는, 특히 System V, socket 함수는 단지 일반 사용자 process의 부분으로 실행하는 라이브러리 함수이다. 어떻게 이 함수들은 kernel에서 protocol stack과 함께 통신을 하는지 상세한 구현은 일반적으로 영향을 끼치지 않는다. 그럼에도 불구하고, 단순함을 위해서, bind와 connect를 포함한 함수에 의해서 process와 kernel 사이에서 전달되는 구조체에 대해서 더 언급할 것이다.

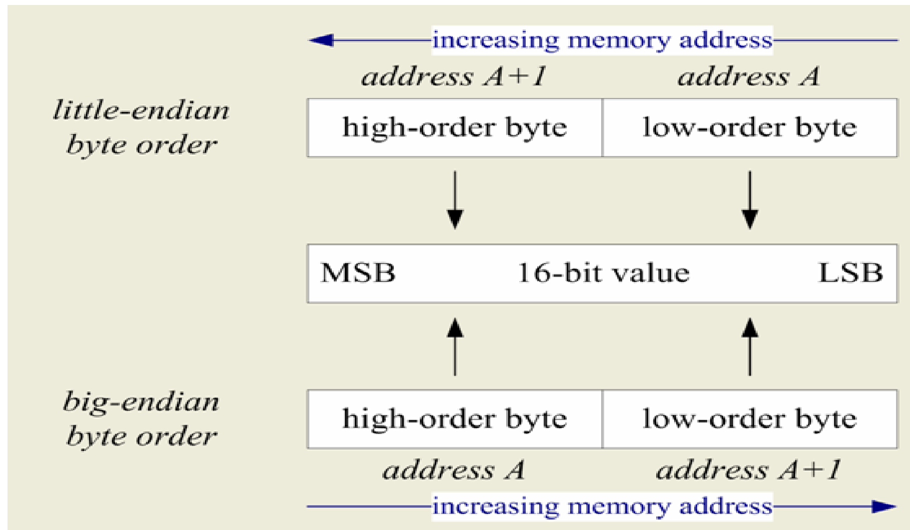
Socket address 구조체의 length field를 위해서 value-result 인자를 사용할 때, 만약 socket address 구조체가 고정 길이라면, kernel에 의해 리턴되는 값은 항상 고정된 길이다: 예를 들면 IPv4 sockaddr_in을 위한 16과 IPv6 sockaddr_in을 위한 28. 그러나 가변 길이 socket address 구조체(예를들면, Unix 계열 sockaddr_un)라면, 리턴되는 값은 구조체의 최대 크기보다 작을 수 있다.

네트워크 프로그래밍에서, value-result 인자의 대부분 일반적인 예는 리턴되는 socket address 구조체의 길이이다. 그러나 이 책에서는 다른 value-result 인자에 대해서 언급할 것이다.

- select 함수를 위한 가운데 세가지 인자 (6장)
- getsockopt 함수를 위한 길이 인자 (7장)

3.3 BYTE ORDERING 함수

2bytes로 구성된 16 bit 정수형을 고려해보자. 메모리에 2bytes를 저장하는 두 가지 방법이 있다: little-endian byte 순으로 잘 알려진, 시작 address에서 내림차순 byte, 또는 big-endian byte로 잘 알려진, 시작 address에서 오름차순 byte. 이 두 가지 형식을 다음 그림에서 볼 것이다.



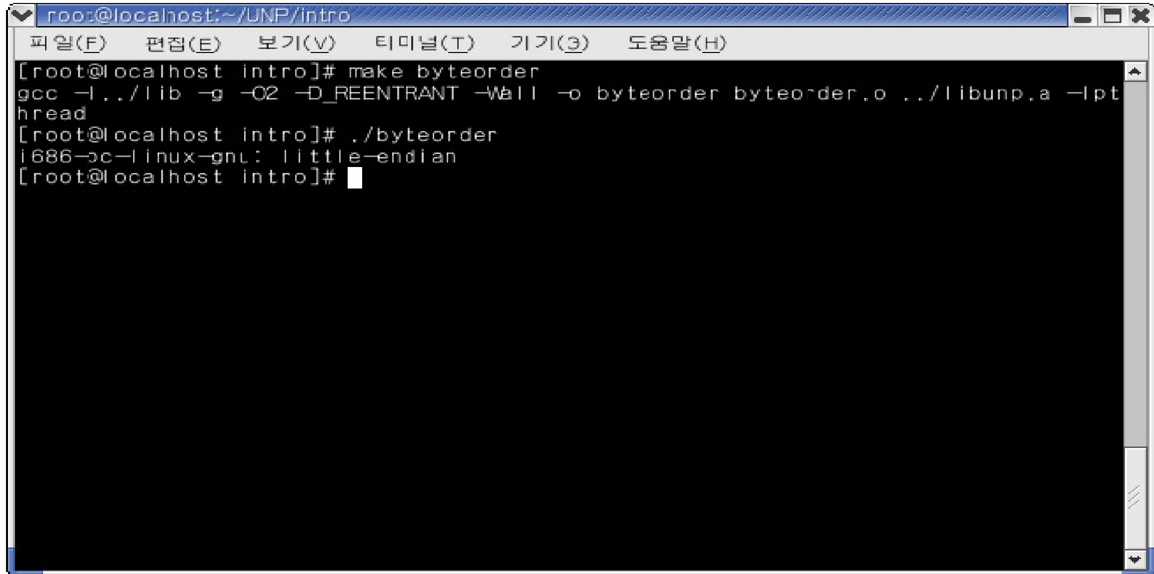
<그림 3.4> Little-endian과 Big-endian

이 그림에서, 위에서 오른쪽에서 왼쪽으로 갈수록 아래쪽에서 왼쪽에서 오른쪽으로 갈수록 메모리 주소가 증가하는 것을 볼 수 있다. 16bit 값의 가장 왼쪽 bit로서 가장 의미 있는 bit(MSB)와 가장 오른쪽 bit로써 가장 의미 없는 bit(LSB)를 볼 것이다. 불행하게도, 이 두 가지 byte 순서 중에 표준은 없다. 그리고 두 가지 형식 모두 사용하는 시스템을 언급한다. host byte order로써 주어진 시스템에 의해 사용되는 byte 순서에 주목한다.

다음 예제에서 보여준 프로그램은 host byte order를 출력한다.

<intro/byteorder.c>

```
---------------------------------------------------------------------------
   1  #include            "lnp.h"
   2  int
   3  main(int argc, char **argv)
   4  {
   5         union {
   6                short  s;
   7                char   c[sizeof(short)];
   8         }un;
   9         un.s = 0x0102;
  10         printf("%s:  ", CPU_VENDOR_OS);
  11         if(sizeof(short) == 2) {
  12                if(un.c[0] == 1 && un.c[1] == 2)
  13                       printf("big-endian\n");
  14                else if (un.c[0] == 2 && un.c[1] == 1)
  15                       printf("little-endian\n");
  16                else
  17                       printf("unknown\n");
  18         } else
  19                printf("sizeof(short) = %d\n", sizeof(short));
  20         exit(0);
  21  }
--------------------------------------------------------------------------
```



<그림 3.5> byteorder.c 실행화면

short integer에 2byte 값 0x0102를 저장하고 두 일관되는 c[0](주소 A)와 c[1](주소 A+1)를 byte order를 결정하기 위해서 살펴본다. 문자열 CPU_VENDOR_OS는 이 책에서 소프트웨어를 형성될 때, CPU종류와 상표, 그리고 OS를 확인하고, GNU autoconf 프로그램에 의해 결정된다. 여기서는 다양한 시스템 상에서 이 프로그램이 작동할 때의 결과를 보여준다.

```
freebsd4 % byteorder
i386-unknown-freebsd4.8: little-endian
macosx % byteorder
powerpc-apple-darwin6.6: big-endian
freebsd5 % byteorder
sparc64-unknown-freebsd5.1: big-endian
aix % byteorder
powerpc-ibm-aix5.1.0.0: big-endian
hpux % byteorder
hppa1.1-hp-hpux11.11: big-endian
linux % byteorder
i586-pc-linux-gnu: little-endian
solaris % byteorder
sparc-sun-solaris2.9: big-endian
```

16-bit 정수형 byte ordering에 대해 이야기 했었다; 명백히, 32-bit 정수형에 동일하게 적용된다. 네트워크 프로그래머로서 이 byte ordering 차이점은 반드시 다뤄야 한다. 왜냐하면 네트워킹 프로토콜은 network byte order를 정해야 한다. 예를들면, TCP 세그먼트에서, 16-bit 포트번호와 32-bit IP 주소가 있다. 전송 프로토콜 스택과 수신 프로토콜 스택은 이 multi-byte 부분의 byte가 전송될 순서가 정해져야 한다. Internet 프로토콜은 이 multibyte 정수형을 위해서 big-endian byte ordering을 사용한다.

규칙에 따라, 구현 시에는 host byte order에서 socket address 구조체 안에 필드를 저장한다. 그리고 프로토콜 헤더에서 이동할 때 네트워크 byte order에서 host byte order로 바뀐다. 이것에 대한 상세한 사항은 우리가 걱정할 필요가 없다. 그러나 예전부터 그리고 POSIX 표준에서는 socket address 구조체에서 특정 필드는 네트워크 byte field로 유지되어야 한다. 그러므로 우리의 관심은 host byte order와 네트워크 byte order 사이에서 바뀌는 것이다. 이 두 가지 byte order 사이에서 변환을 위해 다음 네 가지 함수를 사용할 것이다.

```
#include <netinet/in.h>
uint16_t htons (uint16_t host16bitvalue);
uint32_t htonl (uint32_t host32bitvalue);
                                Both return : value in network byte order
uint16_t ntoh(uint16_t net16bitvalue);
uint32_t ntoh(uint32_t net32bitvalue);
                                   Both return : value in host byte order
```

이 함수의 이름에서, h는 host를 의미하고, n은 network를 의미하고, s는 short를 의미하고, l은 long을 의미한다. "short"와 "long"은 4.2BSD Digital VAX 구현에서부터 사용되었다. 우리는 s를 16-bit 값(TCP 또는 UDP의 port 번호)으로 대신하고, l을 32-bit 값(IPv4 주소)으로 대신한다. 반면에, 64-bit Digital Alpha에서 long 정수형은 64bit를 차지한다. 그러나 htonl 과 ntohl함수는 32-bit 값으로 작동한다.

이 함수들을 사용할 때, host byte order와 네트워크 byte order에 대해서 실질적인 값(big-endian 또는 little-endian)을 고려할 필요는 없다. 우리가 해야 하는 것은 주어진 값을 host와 네트워크 byte order 사이에서 변환하기 위해 적합한 함수를 호출하는 것이다. Internet 프로토콜처럼 같은 byte ordering를 가진 시스템 상에서, 이 네 가지 함수는 null 매크로로 정의 되기도 한다.

3.4 BYTE MANIPULATION 함수

data interpreting없이, data가 C 문자열을 null-terminated되는 것 없이, multibyte 필드상에서 작동하는 두 가지 함수 그룹이 있다. 우리는 socket address 구조체를 다룰 때, 이러한 종류의 함수가 필요하다. 왜냐하면 우리는 IP 주소처럼 0에 대한 byte를 포함할 수도 있고, 그러나 C character 문자열은 없는 필드를 교묘하게 처리할 필요가 있다. <string.h>에 정의된 (string에 대한) str로 시작하는 함수는 null-terminated된 C character 문자열을 처리한다.

함수의 이름이 b(byte에 대한)로 시작하는 첫 번째 함수 그룹은 4.2BSD에서 유래되었고 여전히 socket 함수를 지원하는 몇몇의 시스템에 의해 제공된다. 함수의 이름이 mem(memory에 대한)로 시작하는 두 번째 함수 그룹은 ANSI C 표준에서 유래되었고 여전히 ANSI C 라이브러리를 지원하는 몇몇의 시스템에 의해 제공된다.

비록 이 책에서 사용하는 단 하나 bzero지만, 우리는 우선 Berkeley-derived 함수를 살펴본다. 활용이 존재하는 다른 두 함수, bcopy와 bcmp를 언급할지도 모른다.

```
#include <string.h>

void bzero(void *dest, size_t nbytes);

void bcopy(const void *src, void *dest, size_t n_bytes);

void bcmp(const void *ptr1, const void *ptr2, size_t nbytes);

                               Returns: 0 if equal, nonzero if unequal
```

bzero destination도착지에서 byte에 대한 특정한 수를 0으로 설정한다. 우리는 종종 이 함수를 socket address 구조체를 0으로 초기화 하는데 사용하기도 한다. bcopy는 source에서 destination으로 byte에 대한 특정 숫자를 이동한다. bcmp는 임의의 두 byte string을 비교한다. 만약 두 byte string이 동일하다면 리턴 값은 0이다; 다르면 0이 아니다.

다음 함수들은 ANSI C 함수들이다.

```
#include <string.h>

void *memset(void *dest, int c, size_t len);

void *memcpy(void *dest, const void *src, size_t nbytes);

int memcmp(const void *ptr1, const void *ptr2, size_t nbytes);

                                 Returns: 0 if equal, <0 or >0 if unequal
```

memset은 destination에서 byte에 대한 특정한 수를 c 값으로 설정한다. memcpy는 bcopy와 유사하다. 그러나 두 포인터 인자에 대한 순서가 바뀐다. bcopy는 정확하게 겹치는 필드를 다룬다, 반면에 만약 source와 destination이 겹치면 memcpy의 작동은 정의되어 있지 않다. ANSI C memmove함수는 필드가 겹칠 때 사용되어야 한다.

memcpy를 위한 두 포인터의 순서를 기억하는 하나의 방법은 C에서 대입문처럼 왼쪽에서 오른쪽이 같은 것으로 쓰여진다는 것을 기억하는 것이다.

```
dest = src;
```

memcmp는 두 임의의 byte string을 비교하고 만약 같으면 0을 리턴한다. 만약 같지 않다면, 리턴 값은 0보다 크거나 작다. 첫 번째 다른 ptr1에 의해서 가르켜지는 byte가 다를 경우 0보다 크고, ptr2에 의해 가르켜지는 byte가 다를 경우 0보다 작다. 비교는 두 다른 byte가 unsigned chars 라고 가정하고 이루어진다.

3.5 INET_ATON, INET_ADDR, INET_NTOA 함수

우리는 이 섹션과 다음 섹션에서 주소 변환 함수의 두 가지 분류를 이야기 할 것이다. 그 함수들은 Internet address를 ASCII 문자열(사람이 사용하기 더 선호하는 것)과 네트워크 byte ordered binary 값(socket address 구조체에 저장된 값) 사이에서 변환한다.

1. inet_aton, inet_ntoa, inet_addr은 IPv4 주소를 dotted-decimal 문자열(예를 들면, "206.168.112.96")에서 그것의 32-bit 네트워크 byte ordered binary 값으로 변환한다. 너는 아마도 현존하는 많은 코드에서 이 함수들을 언급할 것이다.

2. 새로운 함수 inet_pton과 inet_ntop는 IPv4와 IPv6 둘 다 다룬다. 우리는 이 두 가지 함수를 다음 섹션에서 이야기한다. 그리고 책을 통해 그 함수들을 사용한다.

```
#include <arpa/inet.h>

int inet_aton(const char *strptr, struct in_addr *addrptr);

                              Returns: 1 if string was valid, 0 on error

in_addr_t inet_addr(const char *strptr);

               Returns: 32-bit binary network byte ordered IPv4 address;

                                                    INADDR_NONE if error

char *inet_ntoa(struct in_addr inaddr);

                                Returns: pointer to dotted-decimal string
```

이 중에서 첫 번째, inet_aton은 strptr에 의해 가르켜지는 C character 문자열을 그것의 32-bit binary 네트워크 byte ordered 값으로 변환한다. 변환된 값은 포인터 addrptr을 통해서 저장된다. 만약 성공하면 1을 리턴하고 실패하면 0을 리턴한다.

inet_addr은 같은 변환을 수행한다, 결과값으로 32-bit binary 네트워크 byte ordered 값을 리턴한다. 이 함수에서 문제점은 모든 232가능한 이진수 값이 유효한 IP addresses라는 것이다. (0.0.0.0부터 255.255.255.255까지), 그러나 그 함수는 에러 발생시 상수 INADDR_NONE (전형적으로 32 one-bits)을 리턴한다. 이것은 dotted-decimal 문자열 255.255.255.255 (broadcast로 제한된 IPv4 주소)이 그것의 binary 값이 함수의 실패를 나타내기 때문에 이 함수에 의해 다뤄질 수 없다는 것을 의미한다.

오늘날, inet_addr은 사용되지 않는다. 그리고 대신에 새로운 코드 inet_aton을 사용한다. 다음 섹션에서 IPv4와 IPv6에서 새로운 함수를 사용하는 것이 더 낫다는 것을 보여줄 것이다.

inet_ntoa함수는 32-bit binary 네트워크 byte ordered IPv4 주소를 그에 상응하는 dotted-decimal 문자열로 바꿔준다. 함수의 리턴 값에 의해 가르켜지는 문자열은 정적인 메모리에 존재한다. 이것은 함수가 재진입하지 않는 것을 의미한다. 이 함수는 구조체 포인터가 아닌, 함수의 인자로써 구조체를 가진다. 실제로 인자로 구조체를 가지는 함수는 드물다. 구조체 포인터로 전달하는 경우가 더 많다.

3.6 INET_PTON, INET_NTOP 함수

이 두 함수는 IPv6에서 새로 나왔다. 그리고 IPv4와 IPv6 주소체계에서 둘 다 작동한다. 우리는 이 두 함수를 책을 통해서 사용한다. 문자 "p"와 "n"은 각각 presentation과 numeric을 상징한다. 주소를 위한 프리젠테이션 형식은 가끔 ASCII 문자열이고 숫자 형식은 socket address 구조체 포함된 이진 값이다.

```
#include <arpa/inet.h>

int inet_pton(int family, const char *strptr, void *addrptr);

Returns: 1 if OK, 0 if input not a valid presentation format, -1 on error

const char *inet_ntop(int family, const void *addrptr, char *strptr, size_t len);

                              Returns: pointer to result if OK, NULL on error
```

두 함수를 위한 family인자는 AF_INET 또는 AF_INET6이다. 만약 family가 지원되지 않는다면, 두 함수는 errno를 EAFNOSUPPORT로 설정해서 에러를 리턴할 것이다.

첫 번째 함수는 strptr에 의해 가르켜지는 문자열을 변환하기 위해 노력한다. 그리고 결과를 포인터 addrptr을 통해 이진수로 저장한다. 만약 성공할 경우 1이 리턴된다. 만약 입력 문자열이 조건으로 지정된 family에 대한 유효하지 않은 표현이라면, 0이 리턴된다.

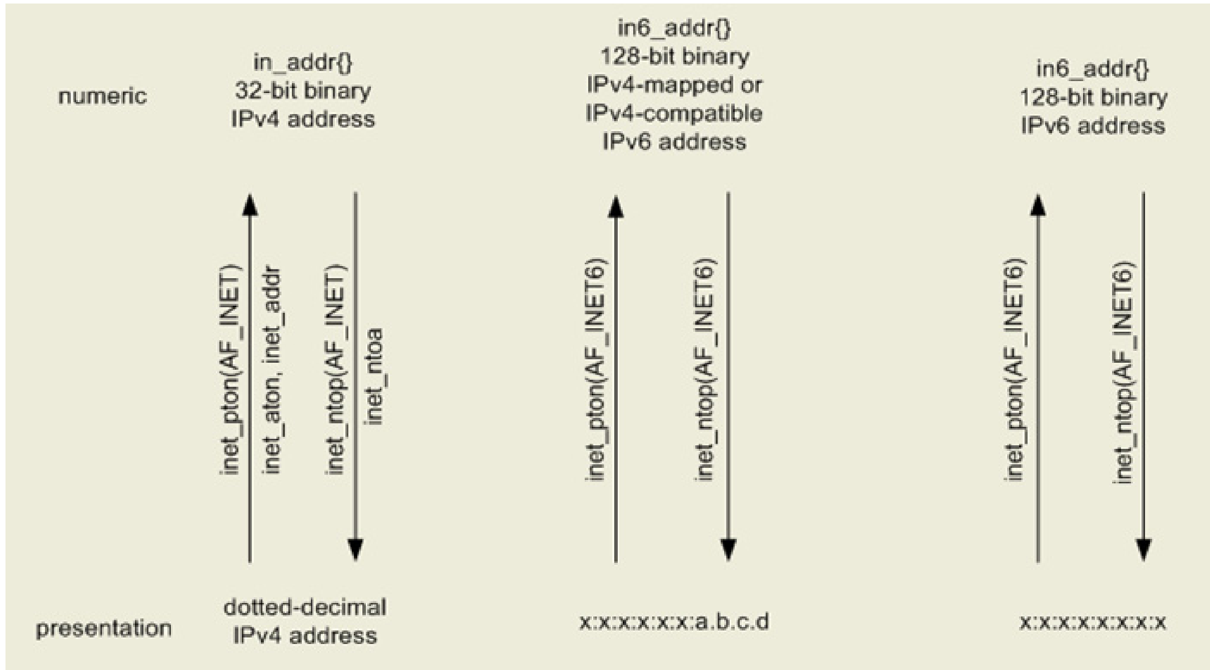
inet_ntop는 반대로 변한다, 숫자형식(addrptr)에서 프리젠테이션 형식(strptr)으로. len 인자는 수신자의 크기이다, 호출자의 버퍼 크기를 초과 하는 것을 방지하기 위해서이다. 이 크기를 지정하는 것을 돕기 위해서, 다음 두 가지 정의가 <netinet/in.h> 헤더에 포함되어 정의 되었다:

```
#define INET_ADDRSTRLEN  16  /* for IPv4 dotted-decimal */
#define INET6_ADDRSTRLEN  46  /* for IPv6 hex string */
```

만약 len이 프리젠테이션 형식을 결과를 보관하기 너무 짧다면, null 종료를 포함하여, null 포인터가 리턴되고 errno가 ENOSPC로 설정된다.

inet_ntop에서 strptr인자는 null 포인터가 될 수 없다. 호출자는 수신자를 위한 메모리를 할당해야하고 그 크기를 정해야 한다. 성공할 경우, 이 포인터는 함수의 리턴 값이다.

다음 그림 지금까지 기술하였던 다섯 함수를 요약하였다.



<그림 3.6> 주소변환 함수 비교

예제

비록 시스템이 아직 IPv6를 지원하지 않더라도, 호출하는 형식을 바꿈으로써 이 새로운 함수들을 사용할 수 있다.

```
foo.sin_addr.s_addr = inet_addr(cp);
```

와 함께

```
inet_pton(AF_INET, cp, &foo.sin_addr);
```

그리고 호출하는 형식을 바꾼다

```
ptr = inet_ntoa(foo.sin_addr);
```

와 함께

```
char str[INET_ADDRSTRLEN];
ptr = inet_ntop(AF_INET, &foo.sin_addr, str, sizeof(str));
```

다음 예제는 단지 IPv4만 지원하는 inet_pton의 간단한 정의를 보여준다. 이와 비슷한, 그다음예제는 단지 IPv4만 지원하는 inet_ntop의 간단한 버전을 보여준다.

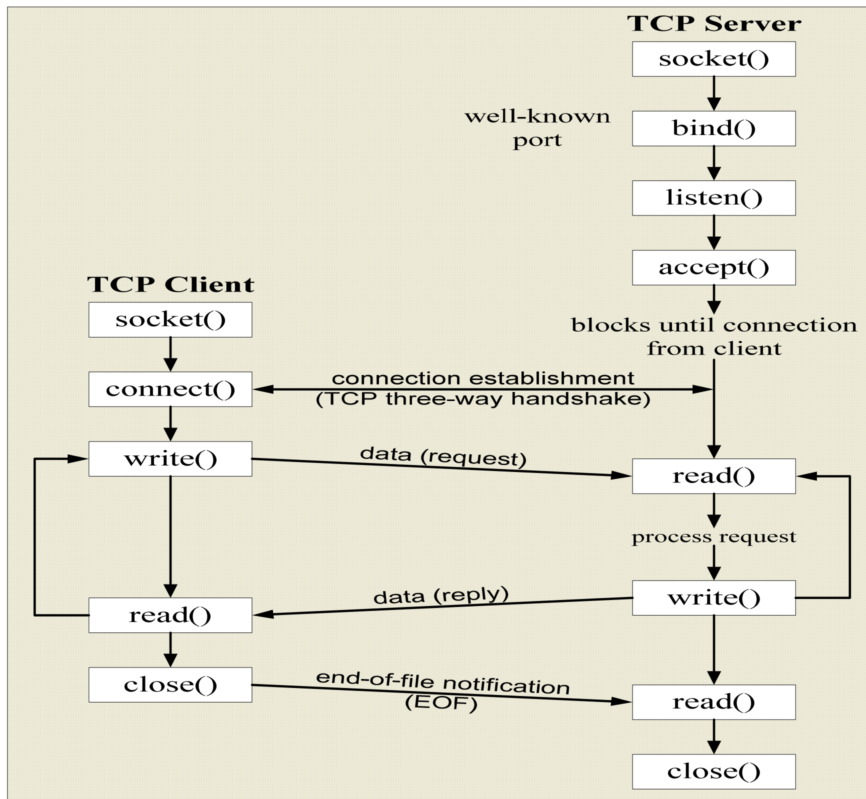
```
<libfree/inet_pton_ipv4.c>
   10 int
   11 inet_pton(int family, const char *strptr, void *addrptr)
   12 {
   13      if (family == AF_INET) {
   14         struct in_addr in_val;
   15         if ( inet_aton(strptr, &in_val)) {
   16            memcpy(addrptr, &in_val, sizeof(struct in_addr));
   17            return (1);
   18         }
   19         return (0);
   20     }
   21     errno = EAFNOSUPPORT;
   22     return (-1);
   23 }
```

```
<libfree/inet_ntop_ipv4.c>
   8  const char *
   9  inet_ntop (int family, const void *addrptr, char *strptr, size_t
      len)
   10 {
   11      const u_char *p = (const u_char *) addrptr;
   12      if (family == AF_INET) {
   13         char temp[INET_ADDRSTRLEN];
   14         snprintf(temp, sizeof(temp), "%d.%d.%d.%d", p[0], p[1], p[2],
      p[3]);
   15         if (strlen(temp) >= len) {
   16             errno = ENOSPC;
   17             return (NULL);
   18         }
   19         strcpy(strptr, temp);
   20         return (strptr);
   21      }
   22      errno = EAFNOSUPPORT;
   23      return (NULL);
   24 }
```

4. TCP Sockets

이번 장에서는 TCP client와 server를 구현하기 위하여 필요한 기본적인 socket functions에 대하여 설명한다. 먼저, 사용할 모든 기본적인 소켓 함수를 설명한 후에 다음 장에서 클라이언트와 서버를 개발한다. 앞으로 클라이언트와 서버를 계속 확장하면서 함께 할 것이다. 또한 동시에 같은 서버에 여러 개의 client가 접속해도 concurrency를 제공하는 유닉스의 보편적인 기법인 다중 접속 서버(concurrent servers)에 대해서도 서술한다. 각각의 client connection마다 server는 전용 프로세스를 fork한다. 이 장에서는, fork를 사용한 one-process-per-client model만 고려한다.

그림 4.1은 TCP client 와 server 사이에서 발생하는 전형적인 시나리오의 모습이다. 처음, server가 시작되고, 얼마 후 server에 연결하는 client가 시작된다. Client는 server에게 request를 전송한다고 가정하면, server는 그 request를 처리하고, client에게 request에 대한 응답을 전송한다. 이것은 client가 server에게 연결을 종료하기 위해 end-of-file로 연결의 끝을 알리는 표시를 전송하여 연결의 한쪽 끝을 닫을 때까지 계속 된다. server는 그 연결을 종료하고 새로운 client의 연결을 기다리거나 종료한다.



<그림 4.1> TCP 소켓함수 시나리오

4.1 SOCKET 함수

```
#include <sys/socket.h>

int socket (int family, int type, int protocol);

Returns: non-negative descriptor if OK, -1 on error
```

Network I/O를 수행하기 위해, 프로세스는 먼저 원하는 통신 프로토콜형 (IPv4, IPv6, 유닉스 영역의 프로토콜 등)을 명시하는 socket 함수를 호출해야 한다. Family는 프로토콜 군을 규정하는데, 표 4.1의 상수 중에 하나를 택한다. 소켓 타입은 표 4.2의 상수 중 하나를 택한다. 보통 소켓 함수의 프로토콜 argument는 그림 4.4에서 규정된 프로토콜 type으로 할당 되어야 하거나 주어진 family와 type의 조합에 대한 시스템의 default로 0으로 정해진다.

<표 4.1> Protocol family 상수

Family	Description
AF_INET	IPv4 protocol
AF_INET6	IPv6 protocol
AF_LOCAL	Unix domain protocols

<표 4.2> 소켓 타입

Type	Description
SOCK_STREAM	Stream socket
SOCK_DGRAM	Datagram socket
SOCK_SEQPACKET	Sequenced packet socket
SOCK_RAW	Raw socket

소켓의 family와 type의 모든 조합이 허용되지는 않는다. 표 4.3은 허용되는 조합과 이들이 실제로 선택하는 프로토콜을 보여준다. Yes로 표시된 칸은 허용은 되지만 빈 칸은 지원되지 않는다.

<표 4.3> socket 함수를 위한 family와 type의 조합

	AF_INET	AF_INET6	AF_LOCAL
SOCK_STREAM	TCP|SCTP	TCP|SCTP	Yes
SOCK_DGRAM	UDP	UDP	Yes
SOCK_SEQPACKET	SCTP	SCTP	Yes
SOCK_RAW	IPv4	IPv6	

첫 인수로써 AF_xxx 대신에 PF_xxx 상수를 볼 수 있다. AF_LOCAL 대신에 AF_UNIX를 사용하기도 한다. SOCK_SEQPACKET는 SCTP만 지원하며, TCP는 SOCK_STREAM 소켓만을 지원한다.

Socket 함수가 성공하면, file descriptor와 유사한 작은 양의정수 값을 반환한다. 이를 socket descriptor 또는 sockfd라고 부른다. 이 socket descriptor를 얻기 위해, protocol family(IPv4, IPv6, 또는 Unix)와 socket type(stream, datagram, 또는 raw)을 지정해야 한다.

AF_xxx vs PF_xxx

AF_ prefix는 "address family"에 대한 약자이고, PF_ prefix는 "protocol_family"의 약자이다. 역사적으로 볼 때, 인터넷에서는 single protocol family가 multiple protocol family를 지원할 수도 있고, PF_값은 소켓을 만드는데 AF_값은 소켓 주소 구조를 만드는데 사용하였다. 그러나 실제로 어떤 프로토콜도 multiple address family를 지원한 적이 없으며, <sys/socket.h> 헤더에는 어느 프로토콜에 대한 PF_값은 그 프로토콜에 대한 AF_값으로 정의되어 있다. 이런 등식이 항상 맞는다고는 할 수 없지만, 만일 누군가가 현존하는 프로토콜에 대해서 이를 바꾸려고 한다면, 거의 모든 프로그램을 고쳐야 할 것이다. 주로 socket을 호출하는 데서 PF_를 보게 되지만, 이 책에서는 상수만을 사용한다. POSIX에서 socket의 첫 인수는 PF_값으로 그리고 AF_ 값은 socket address structure에서 사용한다고 명시하고 있다.

4.2 CONNECT 함수

Connect 함수는 TCP client가 TCP server와 연결을 설정하기 위해 client에 의해서 사용된다.

```
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);

Returns: 0 if OK, -1 on error
```

sockfd는 socket 함수에 의해 반환되는 socket descriptor이다. 두 번째와 세 번째 parameter는 section3.3에서 설명한 socket address structure와 그것의 크기를 나타내는 포인터이다. Socket address structure는 server의 IP 주소와 port 번호를 확실하게 해야 한다.

Client는 connect()를 호출하기 전에 bind() 함수를 호출할 필요는 없다. 커널은 필요하다면 ephemeral port와 source IP 주소를 선택한다.

TCP socket의 경우, connect 함수는 TCP의 three-way handshake를 초기화한다. 이 함수는 연결이 성립하거나 에러가 발생하였을 경우 반환한다. 다음과 같은 여러 가지 오류가 가능하다.

1) client의 TCP가 SYN 세그먼트에 대한 응답을 받지 못하면 ETIMEDOUT을 반환한다. 예를 들면, connect를 호출하면 하나의 SYN을 보내고, 6초 후에 다시 하나를, 그리고 나서 24초 후에 다시 하나를 보낸다. 첫 SYN 이후 75초 동안 응답이 없으면 오류가 발생한 것이다. 어떤 시스템에서는 이 시간 만료를 조정할 수가 있다.

2) Client의 SYN에 대한 server의 응답이 reset(RST)이면, server 호스트에서는 명시한 포트에서 연결을 기다리고 있는 프로세스가 없다는 것이다. (즉, server 프로세스가 아마도 수행중이 아니다.) 이것은 강한 오류(hard error)로써 RST를 수신하자마자 ECONNREFUSED인 오류를 client에게 보낸다.

   RST는 무엇인가 잘못되었을 때 TCP에 의해 전송되는 TCP 세그먼트의 타입이다. RST를 발생시키는 세가지 조건으로 듣고 있는 서버가 없는 포트에 SYN이 도착하거나, TCP가 존재하는 연결을 폐기하기를 원할 때, TCP가 존재하지 않는 연결에 대한 세그먼트를 수신할 때이다.

3) client의 SYN에 대해서 중간 라우터가 "destination unreachable"이라는 ICMP를 보내오면, 이는 약한 오류(soft error)라고 생각한다. Client의 커널은 이 메시지를 보존하고 앞에서 설명한 대로 지속적으로 SYN을 보낸다. 일정 시간 동안 응답을 받지 못하면(예: 75초), 보존한 ICMP 오류인 EHOSTUNREACH 또는 ENETUNREACH를 프로세스에게 돌려준다.

   초기 시스템에서는 "destination unreachable"이라는 ICMP를 받으면 연결 설정 시도를 중단했다. 이렇게 하는 것은 잘못된 것으로 이 ICMP 오류는 일시적인 것이다. 예를 들면, 경로배정 문제에 기인한 상황에 의한 것으로 15초 후에는 바로 잡힐 수도 있다.

1장의 간단한 client에서 이런 여러 가지 오류 조건을 볼 수 있다. 먼저 daytime server를 수행하고 있는 local host(127.0.0.1)를 명시하여 정상적인 출력을 본다.

```
solaris % daytimetcpcli 127.0.0.1
Sun Jul 27 22:01:51 2003
```

다른 형식으로 돌아오는 응답을 보기위해, 다른 장비의 IP 주소를 명시해보자.

```
solaris % daytimetcpcli 192.6.38.100
Sun Jul 27 22:04:59 PDT 2003
```

다음으로, local subnet의 IP 주소(192.168.1/24)와 존재하지 않는 host ID(100)을 명시한다. 다시 말하면, subnet에는 host ID 100의 host가 없으므로 client의 host가 ARP를 보내더라도(그 host에게 하드웨어 주소를 보내라는 요청), ARP 응답을 받지 못할 것이다.

```
solaris % daytimetcpcli 192.168.1.100
connect error: Connection timed out
```

Connect에 대한 시간 만료로 오류를 알게 된다. 참고로 err_sys 함수가 ETIMEDOUT를 사람이 읽을 수 있는 문장으로 인쇄한다.

다음 예제에서는 daytime server가 없는 host(local router)를 명시한다.

```
solaris % daytimetcpcli 192.168.1.5
connect error: Connection refused
```

서버는 즉시 RST를 보내올 것이다.

마지막 예제로써 인터넷에 없는 IP주소를 명시한다. Tcpdump의 패킷으로 6 hop 만큼 떨어진 라우터가 host에 도달 불가라는 ICMP를 보내 왔음을 알 수가 있다.

```
solaris % daytimetcpcli 192.3.4.5
connect error: No route to host
```

ETIMEDOUT 오류처럼, 이 예제에서 connect는 규정된 시간만큼 기다린 연후 EHOSTUNREACH 오류를 돌려준다.

TCP 상태 전이도에 관하여, connect는 CLOSED 상태(socket 함수로 시작한 소켓의 첫 상태)에서 SYN_SENT 상태로 옮기고, 성공하면 다시 ESTABLISHED 상태로 옮긴다. Connect가 실패하면, 소켓은 사용할 수가 없으므로 닫혀야 한다. 그렇지 않으면 그 소켓에 connect를 다시 호출할 수가 없다. connect가 반복문안에서 어떤 호스트에 있는 IP주소를 성공할 때까지 하나씩 시도할 때는, connect가 실패 할 때 마다 socket descriptor를 닫고 다시 호출해야 한다는 것을 알 수 있다.

4.3 BIND 함수

Bind 함수는 local protocol address를 socket에 할당한다. 프로토콜 주소는 32bit IPv4 주소 또는 128bit IPv6 주소와 16bit TCP 또는 UDP 포트 번호와의 결합을 말한다.

```
#include <sys/socket.h>
int bind (int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);
Returns: 0 if OK,-1 on error
```

두 번째 인자는 프로토콜에 의존적인 주소를 가리키는 포인터이고, 세 번째 인자는 이 주소 구조체의 크기이다. TCP에서, bind를 호출하려면 포트 번호 또는 IP 주소 또는 이 둘 모두 또는 이 둘 모두가 아닌 것으로 명시해야 한다.

서버는 시작할 때 well-known port를 bind 한다. TCP client나 server가 포트번호를 지정하지 않으면, 커널이 connect 또는 listen 함수가 호출될 때 socket에 대한 ephemeral port를 임의로 선택한다. 응용에서 포트번호를 지정하지 않으면 TCP client가 커널로 하여금 ephemeral port를 선택

하도록 하는 것은 당연하다. 그러나, TCP server가 커널로 하여금 ephemeral port를 선택하게 하는 것은 드문 일인데, 왜냐하면 server는 well-known 포트로 알려져 있기 때문이다.

이 규칙에서 예외는 RPC(Remote Procedure Call) server이다. 이 server는 토산 listening socket에 대해 ephemeral port를 커널이 택하게 하는데, 왜냐하면 이 포트는 RPC포트 변환기와 함께 등록되기 때문이다. client는 server에 connect하기 전에 포트 변환기를 통해서 ephemeral port를 알게 된다. 이는 UDP를 사용하는 RPC server에도 해당된다.

프로세스는 자신의 socket에 특정한 IP주소를 bind 할 수 있다. IP주소는 호스트의 server에 속해야 한다. TCP client에 있어서, socket으로 보낼 IP 세그먼트에서 사용할 출발지 IP 주소를 부여하는 것이다. TCP server에 있어서, 그 IP 주소로 향하는 클라이언트의 세그먼트만 socket이 받도록 한다.

일반적으로 TCP client는 자신의 socket에 IP 주소를 bind 하지 않는다. 커널은 socket이 연결될 때에 사용되는 밖으로 나가는 interface에 근거해서 IP 주소를 선택하는데 source, 이 접속 역시 서버에 이르는 경로에 근거를 둔다. TCP server가 자신의 socket에 IP 주소를 bind 하지 않으면, 커널은 client의 SYN에 있는 destination IP 주소를 서버의 source IP 주소로 사용한다.

앞에서 말한 대로, bind를 호출하려면 IP 주소 또는 포트 번호를 명시해야 한다. 표 4.4에서 원하는 결과에 따른 sin_addr과 sin_port 또는 sin6_addr과 sin6_port 값을 요약하여 보여준다.

<표 4.4> bind() 함수에 대한 IP 주소와 포트를 지정 예제

프로세스에서의 지정 예제		결과
IP 주소	port	
Wildcard	0	커널에서 IP 주소와 port 지정
Wildcard	nonzero	커널에서 IP 주소 지정, 프로세스에서 port 지정
Local IP address	0	프로세스에서 IP 주소 지정, 커널에서 port 지정
Local IP address	nonzero	프로세스에서 IP 주소 및 port 지정

포트 번호를 0으로 명시하면, bind가 호출될 때에 커널이 ephemeral port를 선택한다. 하지만 wildcard IP 주소(임의의 IP 주소)를 명시하면 커널은 socket이 연결되거나(TCP) datagram이 socket으로 보내질 때까지 local IP 주소를 선택하지 않는다. IPv4에서는 상수인 INADDR_ANY로써 wildcard IP 주소(임의의 주소)를 명시한다. 이 상수의 값은 보통 0이다. 이것이 커널로 하여금 IP 주소를 선택하도록 한다.

```
struct sockaddr_in   servaddr;
servaddr.sin_addr.s_addr = htonl (INADDR_ANY);     /* wildcard */
```

IP 주소가 32비트의 간단한 상수로(이 경우는 0) 표시되는 IPv4에서는 이것이 동작하지만, IPv6에서 이 기법을 사용할 수는 없다. 왜냐하면 128bit의 IPv6 주소가 구조 속에 저장되기 때문이다. (C에서 할당문의 오른쪽에 상수 구조를 사용할 수 없다.)

이 문제를 해결하기 위해 다음을 작성한다.

```
struct sockaddr_in6    serv;
serv.sin6_addr = in6addr_any;     /* wildcard */
```

시스템은 변수인 in6addr_any를 할당하고 상수인 IN6ADDR_ANY_INIT로 초기화 한다. <netinet/in.h> 헤더는 in6addr_any에 대한 extern 선언문을 포함한다.

INADDR_ANY(0)의 값은 네트워크 또는 호스트 바이트 순서와 같아서 htonl을 사용할 필요가 없다. <netinet/in.h> 헤더에서 정의한 INADDR_ 상수는 호스트 바이트 순서이기 때문에 이들 상수에 대해서는 htonl()을 사용해야 한다.

Bind에서 생기는 통상적인 오류는 ADDRINUSE("Address already in use")이다. 7절에서 SO_REUSEADDR와 SO_REUSEPORT socket option을 설명할 때에 이에 대해 좀더 설명한다.

4.4 LISTEN 함수

TCP server 만이 listen 함수를 호출한다. socket 함수가 socket을 만들 때 active socket이라고 가정한다. 즉, connect를 호출할 client socket이라고 가정한다. Listen 함수는 연결되지 않은 socket을 passive socket으로 바꾸며, 커널은 socket에 대한 연결 요청을 받아들이게 된다.

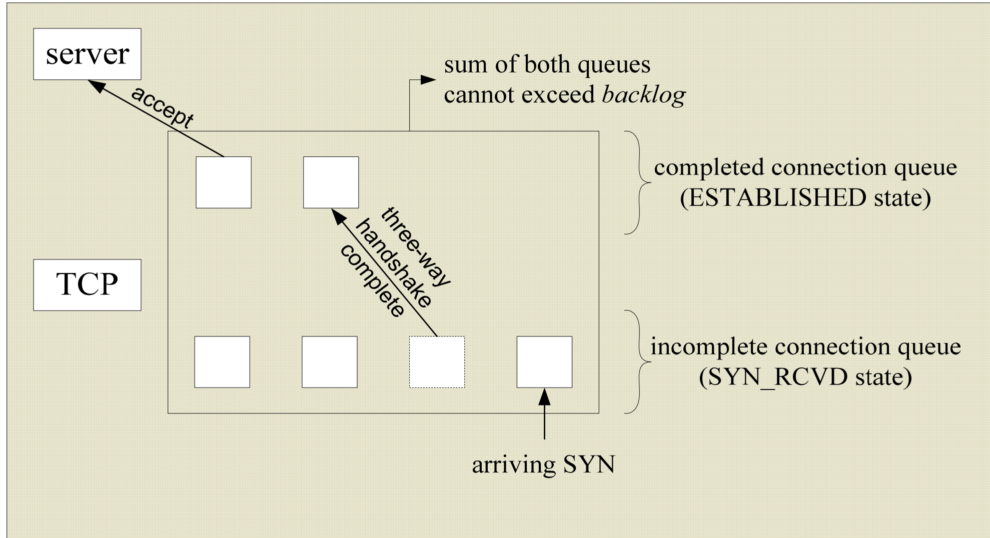
TCP 상태천이도에서 listen의 호출은 socket을 CLOSE 상태에서 LISTEN 상태로 옮기게 한다. 함수의 두 번째 인수는 커널이 socket에 대해 대기시킬 수 있는 최대 연결 개수를 명시한다.

```
#include <sys/socket.h>

#int listen (int sockfd, int backlog);

Returns: 0 if OK, -1 on error
```

정상적으로 이 함수는 bind 함수보다 뒤에 그리고 accept 함수에 앞서서 호출된다. Backlog 인수를 이해하기 위해, listening socket에 대하여 커널은 두 개의 queue를 유지해야 한다.

- 불완전한 connection queue로서 server가 three-way handshake가 완료되기를 기다리는 상대인 client에 도착한 각 SYN에 대해 하나의 entry씩을 가지고 있다. 이런 socket은 SYN_RCVD 상태에 있다.
- 완전한 connection queue로서 TCP three-way handshake 상대인 각 client에 대해 하나의 entry씩을 가지고 있다. 이런 socket은 ESTABLISHED 상태에 있다.

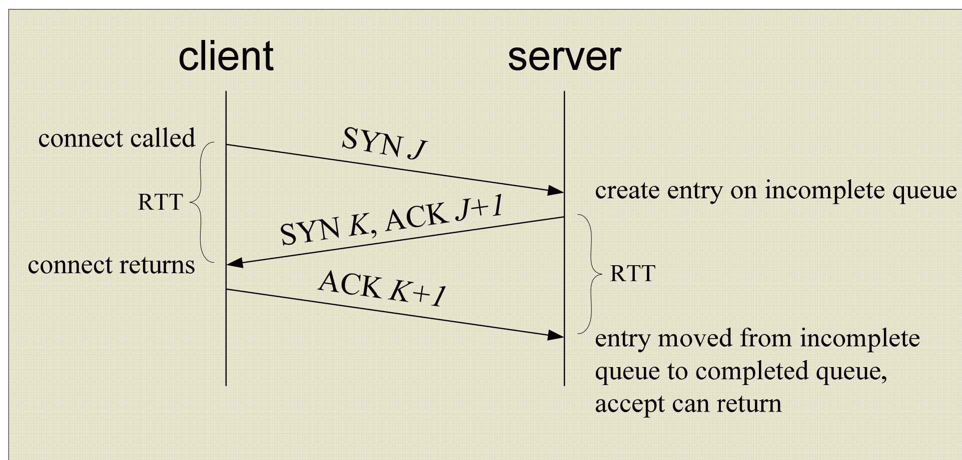
그림 4.2에서 listening socket에 대한 두 개의 queue를 보여준다.



<그림 4.2> TCP listening socket이 관리하는 두 개의 Queue

Entry가 불완전한 queue에서 생성되면, listen socket으로부터의 인자는 새롭게 생성된 연결로 복사된다. 연결 생성 매커니즘은 완전히 자동이다. server 프로세스는 포함되지 않는다.

그림 4.3에서 연결을 설정하는 동안에 교환하는 패킷을 이 두 queue와 함께 보여준다.



<그림 4.3> TCP 연결설정과 listening socket

Client로부터 SYN이 도착하면 TCP는 불완전한 queue에 새로운 entry를 만들고 나서 three-way handshake의 두 번째 세그먼트인 client SYN의 ACK와 함께 server의 SYN으로 응답한다. 이 entry는 three-way handshake의 세 번째 세그먼트가 도착하거나(server의 SYN에 대한 client의 ACK) 또는 이 entry가 시간 만료될 때까지 불완전한 queue에 남아 있다.

three-way handshake가 정상적으로 끝나면 불완전한 queue의 entry는 완전한 queue의 첫 entry가 프로세스에게 돌아가며 만일 queue가 비워 있으면 완전한 queue에 entry가 생길 때까지 프로세스는 sleep 상태에 놓이게 된다.

이 두 queue를 다루는 데 있어서 고려해야 할 것은 다음과 같다.

- listen 함수의 인수인 backlog는 두 queue의 합에 대한 최대 값을 규정한다.
- backlog를 0으로 명시하면 안 된다. 왜냐하면 구현에 따라 다르게 해석할 여지가 있기 때문이다. 어떤 구현에서는 하나의 연결을 허용하고 다른 곳에서는 하나도 허용하지 않는다. 만일 listening socket에 client가 연결하기를 원하지 않는다면 listening socket을 닫는다.
- 역사적으로, sample code에서 backlog는 항상 5였다. Busy server도 하루에 몇 백 건의 연결만 처리했던 1980년대에는 이 값이 적절했다. 그러나 웹의 팽창으로 Busy server는 하루에 수백만 건의 연결을 처리하게 되었기에 이 작은 값은 쓸모가 없어졌다. busy HTTP server는 엄청나게 큰 backlog를 규정해야 하므로 새로운 커널은 이를 지원해야 한다. 요즘 많은 시스템에서 관리자가 backlog를 위한 최대 값을 수정할 수 있도록 한다.

문제는, 5가 부적절하므로 응용에서 backlog 값으로 얼마를 규정해야 하느냐이다. 이에 대한 해답은 쉽지가 않다. HTTP server는 큰 값으로 규정해야 하는데 만일 그 값을 프로그램에서 상수로 규정하면 그 값을 크게 할 때는 다시 server를 컴파일 해야 하다. 다른 방법으로 기본 값을 정한 연후에 command-line의 option이나 기본 값을 덮어버릴 환경 변수를 허용하는 것이다. 커널은 지원할 수 있는 값보다 큰 값은 자체에서 잘라버리므로 커널이 지원하는 값보다 큰 값을 규정하더라도 오류가 발생하지 않으므로 별 문제가 없다.

이 문제의 간단한 해결책은 다음 예제코드처럼 listen 함수의 wrapper function을 수정하는 것이다. 다음 코드에서 환경 변수로써 LISTENQ를 지정된 값으로 대체하도록 한다.

<lib/wrapsock.c>

```
------------------------------------------------------------------------
137 void
138 Listen (int fd, int backlog)
139 {
140     char    *ptr;

141         /* can override 2nd argument with environment variable */
142     if ( (ptr = getenv("LISTENQ")) != NULL)
143         backlog = atoi (ptr);

144     if (listen (fd, backlog) < 0)
145         err_sys ("listen error");
146 }
------------------------------------------------------------------------
```

예전의 Manual과 책에서 연결을 정해진 수만큼 대기시키는 이유는 accept 호출과 다음 호출 동안 server 프로세스가 바쁜 경우를 대비하기 위해서라고 한다. 이는 완전한 queue가 불완전한 것보다 많은 entry를 가져야 한다는 뜻이다. 이번에도 busy Web server에서 이것이 틀렸다는 것을 알 수 있다. 큰 backlog를 규정하는 이유는 client의 SYN이 도착하는 대로 three-way handshake가 완료되기를 기다리는 불완전한 연결 queue가 증가할 수 있도록 하는 것이다.

client의 SYN이 도착할 때에 queue가 다 차있으면, 이 SYN을 무시하고 RST는 보내지 않는다. 이는 일시적인 현상으로 client의 TCP는 SYN을 재전송할 것이고 곧 queue의 공간을 발견하게 될 것이기 때문이다. 만일 server TCP가 RST를 보내게 되면, client의 connect는 즉시 오류를 돌려주게 될 것이고 이로 인해 TCP가 재전송을 하는 대신에 응용에서 이런 상황을 처리해야 한다. client는 SYN에 대한 응답이 RST이면 "there is a server at this port but its queues are full."을 구분할 수도 없기 때문이다.

three-way handshake가 이루어진 연후에 그러나 server가 accept를 호출하기 전에 도착한 데이터는 server TCP가 연결된 socket의 수신용 버퍼가 찰 때까지 그곳에 대기시킨다.

표 4.5는 여러 운영체제와 각기 다른 backlog 인수 값에 대해 실제 대기하고 있는 연결 개수를 보여준다. 7개의 운영체제에서 backlog의 다양한 의미를 5개의 열로 보여주고 있다.

<표 4.5> OS별 backlog 값과 실제 대기되는 연결 수

backlog	실제 대기되는 연결의 최대 수				
	MacOS	Linux	HP-UX	FreeBSD	Solaris
0	1	3	1	1	1
1	2	4	1	2	2
2	3	5	3	3	4
3	4	6	4	4	5
4	5	7	6	5	6
5	7	8	7	6	8
6	8	9	9	7	10
7	10	10	10	8	11
8	11	11	12	9	13
9	13	12	13	10	14
10	14	13	15	11	16
11	16	14	16	12	17
12	17	15	18	13	19
13	19	16	19	14	20
14	20	17	21	15	22

4.5 ACCEPT 함수

accept는 TCP server가 호출하는데 완전한 연결 queue의 맨 앞에 있는 완전한 연결을 돌려준다. 완전한 연결 queue가 비워 있으면 프로세스는 sleep상태에 놓이게 된다.

```
#include <sys/socket.h>

int accept (int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);

Returns: non-negative descriptor if OK, -1 on error
```

Cliaddr과 addrlen 인수는 연결한 상대 프로세스(client)의 프로토콜 주소를 반환하는 데 사용한다. Addrlen은 value-result argument이다. 호출 전에 *addrlen이 가리키는 정수 값을 cliaddr이 가리키는 socket address structure의 크기로 정하며, 호출에서 돌아올 때는, 커널이 socket address structure에 실제로 넣은 바이트 개수를 이 정수가 가지게 된다.

Accept가 성공하면, 결과 값은 커널이 자동으로 생성한 brand-new descriptor이다. 이 새로운 descriptor는 client와의 TCP 연결을 참조한다. Accept를 설명할 때, accept의 첫 인수를 listen socket이라 부르고, accept가 반환하는 값을 연결 socket이라 부른다. 이 두 socket을 구별하는 것이 중요하다. Server가 평생을 같이 할 하나의 listen socket만을 만드는 것이 정상이다. 그러면 커널은 accept된 client 연결마다 하나의 연결 socket을 생성한다. Server가 어떤 client에 대한 서비스를 끝내면 연결 socket은 닫히게 된다.

이 함수는 세 개의 값을 반환한다. 이들은 새로운 socket descriptor 또는 오류 표시인 정수의 회수 코드와 클라이언트 프로세스의 프로토콜 주소와 이 주소의 크기이다. client의 프로토콜 주소가 반환 값으로 필요하지 않으면, cliaddr과 addrlen을 null pointer로 정하면 된다. 연결 socket은 반복문에서 매번 닫히지만 listen socket은 server가 살아 있을 동안 열려있다. 그리고 client의 identity에 관심이 없으므로 accept의 두 번째와 세 번째 인수가 null pointer임을 알 수 있다.

다음 예제는 TCP Daytime Server의 코드이다.

<daytimetcpsrv1.c>

```
-----------------------------------------------------------------------------
 1 #include    "unp.h" 2
 2 #include    <time.h>
 3 int
 4 main(int argc, char **argv)
 5 {
 6    int    listenfd, connfd;
 7    socklen_t len;
 8    struct sockaddr_in servaddr, cliaddr;
 9    char   buff[MAXLINE];
10    time_t  ticks;
11    listenfd = Socket(AF_INET, SOCK_STREAM, 0);
12    bzero(&servaddr, sizeof(servaddr));
13    servaddr.sin_family = AF_INET;
14    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
15    servaddr.sin_port = htons(13);  /* daytime server */
16    Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));
17    Listen(listenfd, LISTENQ);
18    for ( ; ; ) {
19       len = sizeof(cliaddr);
20       connfd = Accept(listenfd, (SA *) &cliaddr, &len);
21       printf("connection from %s, port %d\n",
22             Inet_ntop(AF_INET, &cliaddr.sin_addr, buff, sizeof(buff)),
23             ntohs(cliaddr.sin_port));
24       ticks = time(NULL);
25       snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
26       Write(connfd, buff, strlen(buff));
27       Close(connfd);
28    }
29 }
-----------------------------------------------------------------------------
```

New declarations

7-8

두 개의 새로운 변수를 정의하는데 이는 value-result 변수인 len과 client의 프로토콜 주소를 갖고 있는 cliaddr이다.

Accept connection and print client's address

19-23

len을 socket address structure의 크기로 초기화하고 cliaddr 구조에 대한 pointer와 len에 대한 pointer를 두 번째와 세 번째 인수로 accept에게 전한다. Socket address structure의 32비트 IP 주소를 점으로 구분된 십진수의 ASCII 문자열로 변환하도록 inet_ntop을 호출하며, 네트워크 바이트 순서로 된 16비트 포트 번호를 호스트 바이트 순서로 변환하도록 ntohs을 호출한다.

새로운 서버를 수행하고 클라이언트를 같은 호스트에서 수행하여 연속해서 두 번 서버에 연결시킨 후 클라이언트로부터 다음 출력을 얻었다.

```
solaris % daytimetcpcli 127.0.0.1
Thu Sep 11 12:44:00 2003
solaris % daytimetcpcli 192.168.1.20
Thu Sep 11 12:44:09 2003
```

처음 서버의 IP 주소를 반환 주소(127.0.0.1)로 명시하고 다음으로 실제 IP 주소(192.168.1.20)로 명시한다. 그러면 서버의 출력은 다음과 같다.

```
solaris # daytimetcpsrv1
connection from 127.0.0.1, port 43388
connection from 192.168.1.20, port 43389
```

클라이언트의 IP 주소에 일어나는 일에 대하여 알아야 한다. Daytime client는 bind를 호출하지 않기 때문에 커널이 사용하는 outgoing interface를 기본으로 하여 source IP 주소를 선택한다고 section 4.4에서 언급하였다. 첫 번째 경우는 커널이 source IP 주소를 반환 주소로 정했고 두 번째 경우는 이더넷 접속의 IP 주소로 정했다. 이 예에서 Solaris 커널이 선택한 ephemeral port는 43388과 43389임을 알 수 있다.

서버의 shell prompt는 관리자용으로 흔히 사용하는 #으로 바뀌었다. 서버는 13인 reserved port를 관리자 특권으로 수행해야 한다. 관리자 특권이 없이 bind를 호출하면 실패한다.

4.6 Fork 및 Exec 함수

다음 절에서 다중 접속 서버(concurrent server)를 작성하는 방법을 설명하기 전에 유닉스의 fork 함수를 설명하겠다. 이 함수는 유닉스에서 새 프로세스를 만드는 유일한 방법이다.

```
#include <unistd.h>
pid_t fork(void);
Returns: 0 in child, process ID of child in parent, -1 on error
```

fork를 수행하여 새로운 프로세스(process)를 생성하고 이를 child process라 한다. 그리고 호출한 프로세스를 부모(parent) 프로세스라 한다. 부모 프로세스는 반환 값으로 새로이 만들어진 프로세스(child)의 프로세스 ID를 취한다. 그래서 return value로 현재의 프로세스가 parent인지 child인지를 안다.

Fork가 child 프로세스에서 parent의 프로세스 ID 대신에 0을 돌려주는 이유는 child에게는 한 parent만 있으므로 언제나 getppid를 호출하여 parent의 프로세스 ID를 얻을 수 있기 때문이다. 반면에 parent는 많은 자식을 가질 수 있고 자식의 프로세스 ID를 알 방법이 없기 때문에 자식 ID를 return value로 갖는다. 부모 프로세스가 모든 자식의 프로세스 ID를 추적하려면 fork return value를 기록해 두어야 한다.

Fork를 호출하기 전에 부모가 연 모든 descriptor를 fork에서 돌아온 이 후의 자식도 함께 사용한다. 네트워크 서버가 이 기능을 사용하는데, 부모가 accept를 호출하고 나서 fork를 호출하는 것이다. 그러면 부모와 자식은 connected socket을 공유하게 된다. 보통 자식이 connected socket에서 읽거나 쓰고 나면 부모가 이를 닫는다. 전형적인 fork의 용도는 두 가지이다.

① 프로세스는 자신의 복사본을 만들어서 각기 다른 일을 처리하게 한다. 전형적으로 네트워크 서버가 이렇게 한다. 나중에 많은 예를 보게 될 것이다.

② 프로세스가 다른 프로그램을 수행한다. Fork를 부르는 것이 새 프로세스를 만드는 유일한 길이므로 프로세스는 먼저 fork를 호출하여 자신의 복사본을 만든 후에 둘 중의 하나가(보통 자식 프로세스) exec(다음에 설명한다)를 호출하여 자식을 새 프로그램으로 대체한다. shell 같은 프로그램에서 전형적으로 이렇게 한다.

유닉스에서 디스크에 있는 수행 가능한 프로그램을 수행하는 유일한 방법은 이미 있는 프로세스가 여섯 개의 exec 함수 중에서 하나를 호출하는 것이다. (여섯 개 중 어느 것이냐가 중요하지 않을 때는 일반적으로 그냥 exec함수라고 부른다.) exec는 현 프로세스를 새 프로그램 파일로 대체하며 이 새 프로그램은 보통 main에서 시작한다. 프로세스 ID는 바뀌지 않는다. exec를 호출한 프로세스를 호출 프로세스라 하고 새로이 수행하는 프로그램을 새 프로그램이라 부른다.

여섯 개의 exec 함수가 서로 다른 점은

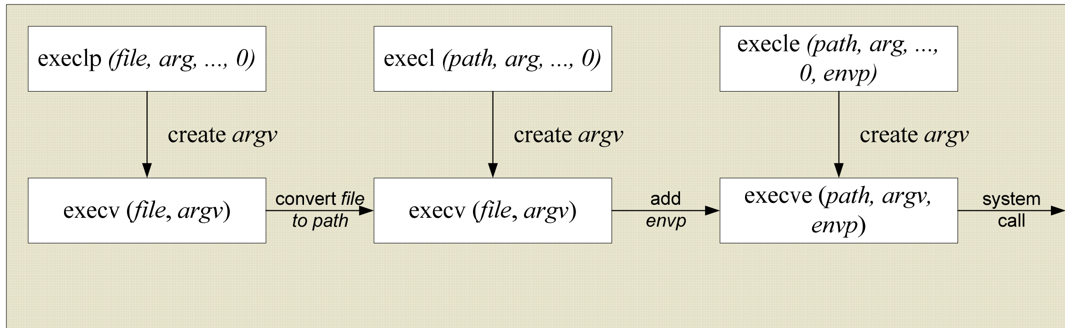
(a) 수행할 프로그램 파일을 파일이름(명)으로 규정하느냐 또는 경로 이름(명)으로 규정하느냐,

(b) 새 프로그램의 인수들이 하나씩 나열되어 있나 또는 pointer 배열을 통하여 참조되고 있나,

(c) 호출 프로세스의 환경이 새 프로그램에 계승되느냐 또는 새 환경이 규정 되느냐, 등이다.

```
#include <unistd.h>
int execl (const char *pathname, const char *arg0, ... /* (char *) 0 */ );
int execv (const char *pathname, char *const argv[]);
int execle (const char *pathname, const char *arg0, ...
/* (char *) 0, char *const envp[] */ );
int execve (const char *pathname, char *const argv[], char *const envp[]);
int execlp (const char *filename, const char *arg0, ... /* (char *) 0 */ );
int execvp (const char *filename, char *const argv[]);
All six return: -1 on error, no return on success
```

이들 함수는 오류 발생시에만 caller에게 돌아간다. 그렇지 않으면 통상 main함수에서 새로운 프로그램이 시작한다. 그림 4.4는 이들 함수의 관계를 보여준다. execve만이 커널이 사용하는 시스템 호출이고 나머지 다섯은 execve를 호출하는 라이브러리 함수이다.



<그림 4.4> EXEC 함수들간의 관계

이들 함수는 다음과 같은 차이를 유의해야 한다.

1. 윗줄에 있는 세 함수는 각 인수 문자열을 exec 함수에 대한 인수로 규정하고, 인수의 수는 유동적이나 빈 pointer로 그 끝을 알려준다. 아랫줄의 세 함수는 인수 문자열을 가리키는 pointer들의 배열인 argv를 가진다. 이 argv 배열에도 그 끝을 알려주는 null pointer가 있어야 한다. 이는 개수를 규정하지 않기 때문이다.

2. 왼쪽 열의 두 함수는 filename argument를 규정한다. 이는 현재의 PATH 환경 변수를 이용하여 경로 이름으로 변환된다. Execlp 또는 execvp의 filename argument에 (/)이 있으면 PATH 변수를 사용하지 않는다. 오른쪽 두 열의 나머지 함수는 적절한 pathname argument를 규정한다.

3. 왼쪽 두 열의 네 함수는 명시적인 environment pointer를 규정하지 않는다. 대신에 외부 변수 environ의 현재 값을 새 프로그램으로 계승될 environment list를 만드는 데 사용한다. 오른쪽 열의 두 함수는 명시적인 environment list를 규정한다. Pointer들의 envp 배열은 null pointer로 끝나야 한다.

4.7 CONCURRENT 서버

앞선 예제의 서버는 iterative server이다. 이는 daytime server같이 단순한 것에는 괜찮다. 그러나 클라이언트의 요청을 서비스하는 데 긴 시간이 걸릴 때는, 서버가 한 클라이언트에 얽매이는 것 보다 동시에 많은 클라이언트를 처리하는 것이 바람직하다. 유닉스에서 다중접속(concurrent)서버를 작성하는 간단한 방법은 각각의 클라이언트를 처리할 자식 프로세스를 fork하는 것이다.

다음 예제는 전형적인 다중접속(concurrent)서버의 틀을 보여주고 있다.

```
pid_t pid;
int   listenfd,  connfd;

listenfd = Socket( ... );

   /* fill in sockaddr_in{} with server's well-known port */
Bind(listenfd, ... );
Listen(listenfd, LISTENQ);

for ( ; ; ) {
   connfd = Accept (listenfd, ... );    /* probably blocks */

   if( (pid = Fork()) == 0) {
      Close(listenfd);    /* child closes listening socket */
      doit(connfd);       /* process the request */
      Close(connfd);      /* done with this client */
      exit(0);            /* child terminates */
   }

   Close(connfd);         /* parent closes connected socket */
}
```

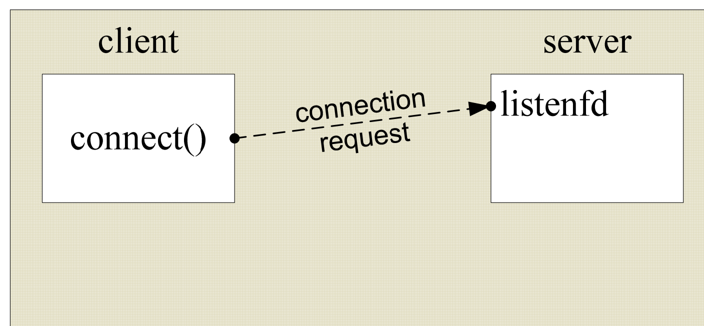
연결이 설정되면 accept에서 반환되고, 서버는 fork를 호출한다. 그러면 자식 프로세스가 클라이언트를 서비스하며(connfd로서 connected socket) 부모 프로세스는 다른 연결을 기다린다. (listenfd로서 listening socket) 자식이 새 클라이언트를 처리하므로 부모는 connected socket을 닫는다.

위 예에서 doit 함수가 무엇인지는 모르지만 클라이언트를 서비스한다고 가정한다. 이 함수에서 반환하면, 자식이 가지고 있는 connected socket을 명시적으로 close한다. 사실 다음 문장이 exit이므로 close는 필요가 없다. 프로세스가 종료하면 커널에 의해 모든 open descriptor는 닫히게 된다. 이 close를 포함하는 것은 개인의 취향에 달려 있다.

2절에서 TCP 소켓에서 close를 호출하면 FIN을 보내게 되고 정상적인 연결 종료 절차가 이어진다고 했다. 왜 상기 예제에서 부모가 호출한 connfd에 대한 close는 클라이언트와의 연결을 끊지

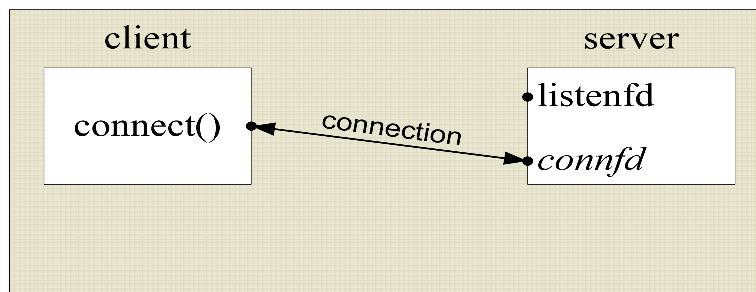
못할까? 이유를 이해하려면 모든 파일이나 소켓은 reference count를 가지고 있음을 알아야 한다. Reference count는 file table entry에 있으며, 파일이나 소켓을 참조하는 open descriptor의 현재 개수를 센다. 예제에서 socket에서 반환한 후, listenfd와 연계된 file table entry에는 reference count의 값이 1로 된다. accept에서 돌아오면, connfd와 연계된 file table entry에는 reference count의 값이 1로 된다. 그러나 fork에서 반환된 후에는 부모와 자식이 두 descriptor를 공유하므로 두 소켓과 연계된 file table entry의 reference count는 2가 된다. 그래서 부모가 connfd를 닫더라도 reference count가 2에서 1로 줄어들 뿐이다. Reference count가 0이 되기 전에는 descriptor가 실제로 닫히는 것이 아니다. 이는 자식이 connfd를 닫을 때에 일어난다.

앞선 예제에서 발생하는 소켓과 연결을 가시화할 수 있다. 먼저 그림 4.5에서 서버가 accept를 호출하여 블록(block)되고 클라이언트로부터 연결 요청이 도착하는 동안의 서버와 클라이언트의 상태를 보여준다.



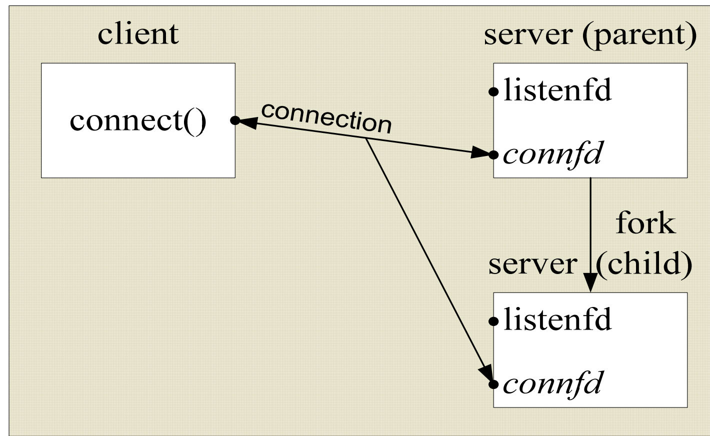
<그림 4.5> Accept() 함수의 반환전 상태

accept에서 반환하는 즉시, 그림 4.6과 같이 된다. 커널이 연결을 허용하고 새 소켓인 connfd를 만든다. 이는 connected socket으로 이 연결을 통하여 데이터를 읽거나 쓸 수가 있다.



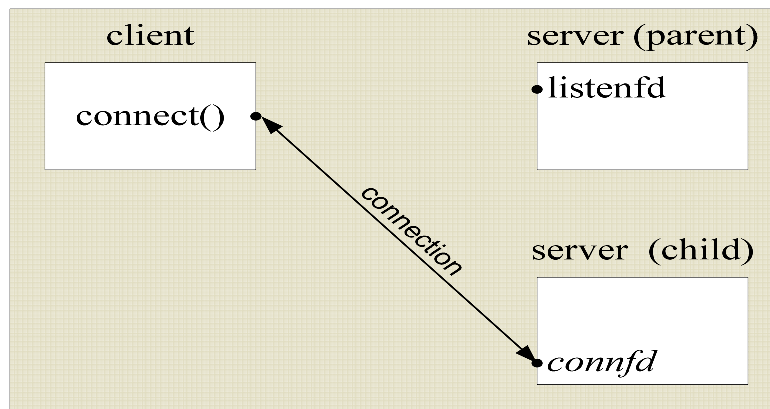
<그림 4.6> Accept() 함수의 반환후 상태

다중 접속(concurrent)서버에서 다음 단계는 fork를 호출하는 것이다. 그림 4.7에서 fork로부터 돌아온 후의 상태를 보여준다.



<그림 4.7> fork() 함수 실행 후 상태

부모와 자식이 두 descriptor인 listenfd와 connfd를 공유함을 알아야 한다. 다음 단계에서 부모는 connected socket을 닫고 자식은 listening socket을 닫는다. 그림 4.8에서 이것을 보여주고 있다.



<그림 4.8> parent와 child 소켓이 분리된 상태

이것이 바람직한 소켓의 마지막 상태이다. 자식이 클라이언트와의 연결을 담당하고 부모는 새로운 클라이언트 연결을 기다리도록 listening socket에서 accept를 다시 호출할 수 있다.

4.8 CLOSE 함수

유닉스의 통상적인 close 함수를 소켓을 닫고 TCP 연결을 종료하는데 사용한다.

```
#include <unistd.h>
int close (int sockfd);
Returns: 0 if OK, -1 on error
```

TCP 소켓에 대한 close의 기본 동작은 소켓을 닫혔다고 표시한 후에 즉시 프로세스로 돌아가는 것이다. 프로세스는 더 이상 socket descriptor를 쓸 수가 없다. 즉, read와 write의 인수로 사용이 불가능 하다. 그러나 TCP는 다른 쪽으로 보낼 데이터가 이미 queue에 들어 있으면 이를 보낸다. 그런 후에 정상적인 TCP 연결 종료 절차를 따른다.

Descriptor Reference Counts

다중접속(concurrent) 서버의 부모 프로세스가 connected socket을 close하면 descriptor에 대한 reference count가 감소한다고 했다. 아직 reference count 값이 0보다 크므로 이 close 호출로 인하여 TCP의 연결종료 절차를 시작하지는 않는다. 이것이 부모와 자식이 공유하는 connected socket에 대한 다중 접속(concurrent)서버의 바람직한 동작이다.

TCP 연결로 FIN을 바로 보내려면, close 대신에 shutdown 함수를 이용할 수 있다 (6절 참조).

accept로 얻은 connected socket을 하나씩 부모가 close하지 않으면 다중접속 서버에서 어떤 일이 일어날지를 알아야 한다. 먼저 부모는 궁극적으로 descriptor의 고갈을 맞게 될 것이다. 이는 어떤 프로세스가 어느 시점에서 열수 있는 descriptor의 개수가 한정되어 있기 때문이다. 그러나 더욱 중요한 것은 어느 클라이언트 연결도 끝나지 않는다는 것이다. 자식이 connected socket을 닫을 때, reference count 값은 2에서 1로 되고 부모가 connected socket을 닫지 않으므로 그대로 1로 남아 있게 된다. 그래서 TCP 연결 종료 절차는 일어나지 않으며 연결은 열린 채로 남게 된다.

4.9 GETSOCKNAME 및 GETPEERNAME 함수

이 두 함수는 소켓과 연계된 local protocol address를 반환하며(getsockname), getpeername은 소켓과 연계된 foreign protocol address를 반환한다.

```
#include <sys/socket.h>
int getsockname(int sockfd, struct sockaddr *localaddr, socklen_t *addrlen);
int getpeername(int sockfd, struct sockaddr *peeraddr, socklen_t *addrlen);
Both return: 0 if OK, -1 on error
```

이 두 함수의 마지막 인수는 "value-result" 인수임을 알아야 한다. 즉 두 함수가 localaddr 또는 peeraddr이 가리키는 socket address structure를 채운다.

이 두 함수는 네트워크 연결의 한쪽 끝의 프로토콜 주소를 IP 주소와 포트 번호의 결합 형태로 돌려준다. 이들 함수와 domain name과는 관계가 없다.

다음과 같은 이유로 이 두 함수가 필요하다.

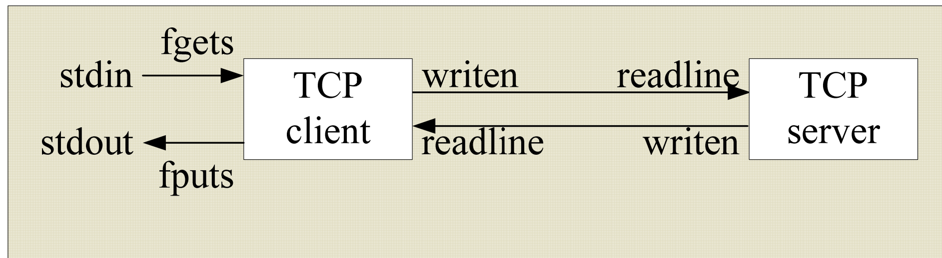
- bind를 호출하지 않는 TCP 클라이언트에서 connect가 성공적으로 끝난 후에는 getsockname으로 local IP 주소와 커널이 연결에 부여한 local 포트 번호를 얻을 수 있다.
- 포트번호 0(커널이 local 포트번호를 고르도록 한다)으로 bind를 호출한 후에는 getsockname으로 부여 받은 local 포트 번호를 얻을 수 있다.
- 소켓의 address family를 getsockname을 호출하여 얻을 수 있다.
- 임의의 IP 주소로 bind한 TCP 서버에서 클라이언트와 연결을 설정하면(accept가 성공하면) 서버에서 getsockname을 호출하여 연결에 부여된 local IP 주소를 얻을 수 있다. 이 호출의 socket descriptor argument는 connected socket에 대한 것이다.

5. TCP CLIENT/SERVER 예제

TCP 클라이언트/서버 예제의 완벽한 구현을 위해 앞 장에서 소개한 기본적인 함수를 이용한다. 단순한 예제는 다음의 과정을 수행하는 에코 서버이다.

1. 클라이언트는 자신의 표준 입력 장치로부터 문자열을 읽어 서버로 보낸다.
2. 서버는 입력으로부터 행을 읽고 다시 클라이언트로 읽은 행을 되돌려 보낸다.
3. 클라이언트는 서버가 되돌려준 행을 읽고 자신의 표준 출력 장치에 쓴다.

그림 5.1은 입출력을 위해 사용하는 함수와 함께 단순한 클라이언트/서버를 그리고 있다.



<그림 5.1> Echo Client/Server 시나리오

위의 그림을 보면 클라이언트와 서버 사이에 두 개의 화살표가 있으나, 이것은 하나의 full-duplex TCP 연결이다. fgets와 fputs 함수는 표준 입출력 라이브러리이다.

에코 서버를 구현하지만, 대부분의 TCP/IP 구현은 TCP와 UDP를 사용하여 그와 같은 서버를 제공한다. 여기서 개발한 클라이언트와 함께 이 서버를 사용한다.

입력을 에코로 돌려보내는 클라이언트/서버는 단순하지만 유용한 네트워크 응용의 예이다. 복잡한 클라이언트/서버를 구현하기 위해 필요한 모든 기본적인 과정을 이 예에서 예시하고 있다. 이 예제를 독자가 자신의 응용에 확장시키기 위하여 해야 할 필요한 일은 자신의 클라이언트로부터 받은 입력을 가지고 서버가 하는 일을 바꾸는 것이다.

이 예제에서는 클라이언트와 서버가 정상적인 방식으로 동작하는 것(행을 쳐 넣고 그것이 되돌아 오는 것을 보자.) 외에 많은 한계 조건을 시험한다.

5.1 TCP Echo Server

다음 예제는 다중 접속(concurrent)서버 프로그램을 보여준다.

<tcpserv01.c>

```
-----------------------------------------------------------------------------
 1 #include     "lnp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5    int    listenfd, connfd;
 6    pid_t   childpid;
 7    socklen_t clilen;
 8    struct sockaddr_in cliaddr, servaddr;
 9    listenfd = Socket (AF_INET, SOCK_STREAM, 0);
10    bzero(&servaddr, sizeof(servaddr));
11    servaddr.sin_family = AF_INET;
12    servaddr.sin_addr.s_addr = htonl (INADDR_ANY);
13    servaddr.sin_port = htons (SERV_PORT);
14    Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));
15    Listen(listenfd, LISTENQ);
16    for ( ; ; )  {
17       clilen = sizeof(cliaddr);
18       connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);
19       if ( (childpid = Fork()) == 0) { /* child process */
20          Close(listenfd);    /* close listening socket */
21          str_echo(connfd);   /* process the request */
22          exit (0);
23       }
24       Close(connfd);         /* parent closes connected socket */
25    }
26 }
-----------------------------------------------------------------------------
```

Create socket, bind server's well-known port

9-15

TCP 소켓을 생성한다. 인터넷 소켓 주소 구조는 임의 주소(INADDR_ANY)와 서버의 well known port로 채운다. Wildcard address를 바인딩함으로써 시스템이 여러 네트워크에 연결된 경우에는 어느 local interface로 오는 연결도 허용하도록 한다. 소켓은 listen에 의해 listening socket으로 바뀐다.

Wait for client connection to complete

17-18 서버는 클라이언트로부터의 연결이 완료되기를 기다리며 accept에서 블록된다.

Concurrent server

19-24

각 클라이언트에 대해 서버는 fork를 호출하여 자식 프로세스를 만들고 자식이 클라이언트를 처리한다. Section 4.8에서 논의한 대로 자식은 listening socket을 닫고 부모는 connected socket을 닫는다. 그런 후 자식은 클라이언트를 처리하기 위하여 str_echo 함수를 (뒤에서 설명) 호출한다.

함수 str_echo는 각 클라이언트를 위해 서버가 해야 할 일을 수행한다. 즉, 클라이언트로부터 행을 읽어 다시 클라이언트에 되돌려 준다.

<str_echo.c>

```
--------------------------------------------------------------------------
 1 #include    "unp.h"

 2 void
 3 str_echo(int sockfd)
 4 {
 5    ssize_t n;
 6    char    buf[MAXLINE];

 7   again:
 8    while ( (n = read(sockfd, buf, MAXLINE) ) > 0)
 9       Writen(sockfd, buf, n);

10    if (n < 0 && errno == EINTR)
11       goto again;
12    else if (n < 0)
13       err_sys("str_echo: read error");
14 }
--------------------------------------------------------------------------
```

Read a buffer and echo the buffer

8-9

소켓으로부터 데이터를 읽고 writen으로 클라이언트에게 되돌려준다. 클라이언트가 연결을 닫으면(the normal scenario), 클라이언트의 FIN을 수신함으로써 자식의 readline은 0을 반환한다. 이로써, str_echo 함수에서 돌아오고, 자식 프로세스는 종료한다.

5.2 TCP ECHO CLIENT

다음 예제는 TCP Echo 클라이언트의 main함수를 보여준다.

```
<tcpcli01.c>
---------------------------------------------------------------------------
 1 #include   "unp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5    int    sockfd;
 6    struct sockaddr_in servaddr;
 7    if (argc != 2)
 8       err_quit("usage: tcpcli <IPaddress>");
 9    sockfd = Socket(AF_INET, SOCK_STREAM, 0);
10    bzero(&servaddr, sizeof(servaddr));
11    servaddr.sin_family = AF_INET;
12    servaddr.sin_port = htons(SERV_PORT);
13    Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
14    Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));
15    str_cli(stdin, sockfd);    /* do it all */
16    exit(0);
17 }
---------------------------------------------------------------------------
```

다음은 상기 클라이언트 예제의 실행화면이다.

```
[LeeDonghwa@localhost tcpcliserv]$ ./tcpcli01 155.230.105.166
hello
hello
```

Create socket, fill in Internet socket address structure

9-13

TCP 소켓을 생성하고, internet socket address structure를 서버의 IP 주소와 포트번호로 채운다. 서버의 IP 주소는 command-line argument를 통해 얻고, 서버의 well known port(SERV_PORT)는 unp.h 헤더에 있다.

Connect to server

14-15

connect로 서버와의 연결을 설정한다. 함수 str_cli()는 클라이언트의 나머지 일을 수행한다.

다음 예제에서 str_cli() 함수는 클라이언트의 일을 반복해서 수행한다. 즉, 표준 입력에서 문자열을 받아 서버로 보내고, 서버로부터의 메아리를 읽고, 표준 출력에 보여준다.

```
<str_cli.c>
--------------------------------------------------------------------------
 1 #include    "unp.h"

 2 void
 3 str_cli(FILE *fp, int sockfd)
 4 {
 5    char    sendline[MAXLINE], recvline[MAXLINE];

 6    while (Fgets(sendline, MAXLINE, fp) != NULL) {

 7        Writen(sockfd, sendline, strlen (sendline));

 8        if (Readline(sockfd, recvline, MAXLINE) == 0)
 9           err_quit("str_cli: server terminated prematurely");

10        Fputs(recvline, stdout);
11    }
12 }
--------------------------------------------------------------------------
```

Read a line, write to server

6-7

fgets으로 문자열을 읽고, writen으로 읽은 것을 서버에게 보낸다.

Read echoed line from server, write to standard output

8-10

readline으로 서버에서 보내준 메아리를 읽고 fput을 사용하여 표준 출력에 쓴다.

Return to main

11-12

fget이 null pointer를 반환하면 반복문은 종료한다. 이는 end-of-file(EOF)에 이르거나 오류를 만날 때 발생한다. Fgets wrapper 함수는 오류를 점검하고 만일 오류가 발생하면 중단하므로 Fgets는 end-of-file을 만날 때만 null pointer를 반환한다.

5.3 정상적인 시작과 종료

Normal Startup

비록 여기서 소개한 TCP 예제는 작지만 (두 개의 main 함수와 str_echo, str_cli, readline, witen을 망라하여 약 150줄), 클라이언트와 서버가 어떻게 시작하고 어떻게 종료하는지를 알려준다. 그리고 무언가 잘못 되었을 때(클라이언트 호스트가 고장 나고, 클라이언트 프로세스에 이상이 발생하고, 네트워크 연결을 잃어버리는 등) 어떤 일이 발생하는지를 이해하는 데 아주 중요하다. 이들 한계 조건(boundary conditions)과 이들의 TCP/IP 프로토콜과의 상호 작용을 이해함으로써, 이들 조건을 처리할 수 있는 클라이언트와 서버를 작성할 수 있다.

먼저 호스트 linux의 background에서 서버를 시작한다.

```
linux % tcpserv01 &
 [1] 17870
```

서버를 시작할 때, socket, bind, listen, accept를 차례로 호출하고 accept 호출에서 블록킹된다. (아직까지 클라이언트는 시작하지 않았다.) 클라이언트를 시작하기 전에, 서버의 listening socket의 상태를 확인하기 위해 netstat 프로그램을 돌린다.

```
linux % netstat -a
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address      Foreign Address     State
tcp       0      0    *:9877              *:*                LISTEN
```

여기에서는 출력의 첫 번째 줄(the heading)과 관심이 있는 줄만 보여준다. 그러나 실제로는 이 명령으로 시스템의 모든 소켓 상태를 볼 수 있어 많은 줄이 출력될 것이다. Listening socket을 보려면 반드시 –a flag를 지정해야 한다.

기대한 대로 출력되었다. 임의(wildcard) local IP 주소와 9877의 local 포트를 가진 하나의 소켓이 LISTEN 상태에 있다. Netstat는 IP 주소 0(INADDR_ANY, the wildcard) 또는 포트 0에 대해서는 별표(asterisk)를 프린트한다.

다음으로 같은 호스트에서 클라이언트를 시작하면서 서버의 IP 주소는 127.0.0.1(loopback)로 지정한다. 또한 이 주소를 서버의 normal(non-loopback) IP 주소로 지정할 수도 있다.

```
linux % tcpcli01 127.0.0.1
```

클라이언트는 socket과 connect를 호출하는데, 후자는 TCP의 three-way handshake를 시작하게 한다. Three-way handshake가 완료되면, 클라이언트에서는 connect가 돌아오고 서버에서는 accept가 돌아온다. 연결이 설정되면 다음 단계를 거치게 된다.

1. 클라이언트는 str_cli를 호출한다. 아직 입력을 하지 않았기 때문에 fgets에서 블록될 것이다.
2. 서버에서 accept가 반환될 때, 서버는 fork를 호출하고 자식은 str_echo를 호출한다. 이 함수는 readline을 호출하고, readline은 다시 read를 호출하는데, read는 클라이언트에서 보낸 문자열을 기다리면서 블록된다.
3. 다른 한편, 서버 부모는 다시 accept를 호출하여 다음 클라이언트 연결 요청을 기다리면서 블록 된다.

클라이언트와 서버 부모와 서버 자식의 세 프로세스가 있는데 모두가 블록 되어 있다.

Three_way handshake가 완료 될 때, 의도적으로 먼저 클라이언트의 처리 과정을, 다음으로 서버의 처리 과정을 나열했다. Three-way handshake에서 두 번째 세그먼트를 클라이언트가 수신하면 connect에서 반환하고, 세 번째 세그먼트를 서버가 수신할 때까지 accept에서 반환하지 않는다. 즉 accept는 connect가 반환한 후 RTT의 약 반만큼 뒤에 반환한다.

의도적으로 같은 호스트에서 클라이언트와 서버를 수행하는데, 그 이유는 클라이언트-서버 응용을 가장 쉽게 실험해 볼 수 있기 때문이다. 같은 호스트에서 클라이언트와 서버가 수행하므로 netstat을 통해 TCP 연결에 해당하는 두 개의 줄이 추가되어 있다.

```
linux % netstat -a
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address          Foreign Address          State
tcp        0      0 local host:9877       localhost:42758          ESTABLISHED
tcp        0      0 local host:42758      localhost:9877           ESTABLISHED
tcp        0      0 *:9877                *:*                      LISTEN
```

첫 번째 ESTABLISHED 줄은 local 포트 번호가 9877이므로 서버 자식의 소켓에 해당한다. 두 번째 ESTABLISHED 줄은 local 포트 번호가 42758이므로 클라이언트의 소켓이다. 만약 클라이언트와 서버를 서로 다른 호스트에서 수행하면 클라이언트 호스트는 클라이언트 소켓만을 보여주고 서버 호스트는 두 개의 서버 소켓만 보여줄 것이다.

이들 프로세스의 상태와 관계를 점검하려면 ps 명령을 사용할 수 있다.

```
linux % ps -t pts/6 -o pid,ppid,tty,stat,args,wchan
  PID  PPID TT       STAT COMMAND          WCHAN
22038 22036 pts/6     S    -bash            wait4
17870 22038 pts/6     S    ./tcpserv01       wait_for_connect
19315 17870 pts/6     S    ./tcpserv01       tcp_data_wait
19314 22038 pts/6     S    ./tcpcli01 127.0  read_chan
```

이 출력은 같은 윈도우에서 클라이언트와 서버를 수행한다. PID와 PPID 열은 부모와 자식과의 관계를 보여준다. 자식의 PPID는 부모의 PID이므로 첫 번째 tcpserv01 줄은 부모이고 두 번째 tcpserv01 줄은 자식이다. 또한 부모의 PPID는 shell이다.

세 개의 네트워크 프로세스 모두의 STAT 열은 "S"인데, 이는 프로세스가 수면상태임을 나타낸다. 프로세스가 수면 상태이면, WCHAN 열은 그 조건을 명시한다. Linux는 프로세스가 accept나 connect에서 블록되어 있으면 wait_for_connect를, 프로세스가 소켓 입력이나 출력에서 블록되어 있으면 tcp_data_wait를, 프로세스가 terminal I/O에서 블록되어 있으면 read_chain을 출력한다. 세 네트워크 프로세스에 대한 WCHAN 값은 의미가 있다.

Normal Termination

연결이 설정된 시점에서 클라이언트에게 입력한 것이 다시 메아리되어 돌아온다.

```
linux % tcpcli01 127.0.0.1
hello, world
hello, world
good bye
good bye
^D
```

두 줄을 입력하고 그 각각이 echo된 후에, 단말기의 EOF 문자(Control-D)를 쳐 넣으면 클라이언트는 종료된다. 즉시 netstat를 실행하면 다음을 얻는다.

```
linux % netstat -a | grep 9877
tcp        0      0 *:9877              *:*              LISTEN
tcp        0      0 localhost:42758     localhost:9877   TIME_WAIT
```

연결의 클라이언트 측은(local 포트가 42758이므로) TIME_WAIT 상태에 들어가고 listening socket 서버는 여전히 또 다른 클라이언트의 연결을 기다린다.(이번에는 netstat의 출력을 grep로 이관하면 서버의 well-known port만 한 줄씩 인쇄되는데, 이렇게 하면 the heading line도 제거가 된다.)

클라이언트와 서버의 정상적인 종료에 관계된 과정을 따를 수 있다.

1. EOF를 치면, fgets는 null pointer를 반환하고 함수 str_cli()는 반환한다.
2. str_cli가 클라이언트의 main 함수()로 반환되면, 후자는 exit을 호출하며 끝난다.
3. 모든 open descriptors를 닫는 것은 프로세스 종료의 한 부분이므로 클라이언트 소켓을 커널이 닫는다. 이는 서버에게 FIN을 보내게 하며, 서버 TCP는 ACK로써 응답을 할 것이다. 여기까지가 TCP 연결 종료 절차의 첫 번째 반쪽에 해당한다. 이때 서버 소켓은 CLOSE_WAIT 상태에 있고, 클라이언트 소켓은 FIN_WAIT_2 상태에 있다.
4. 서버 TCP가 FIN을 받을 때, 서버 자식은 readline() 호출에서 블록되어 있으므로 readline은 0을 반환하게 된다. 이로써 str_echo함수는 서버 자식의 main으로 돌아가게 된다.
5. 서버 자식은 exit를 호출하여 종료한다.

6. 서버 자식에게 열려있는 모든 descriptors는 닫힌다. 자식에 의해 connected socket이 닫히면 TCP 연결 종료의 마지막 두 세그먼트가 발생한다. 이들은 서버에서 클라이언트로의 FIN과 클라이언트에서의 ACK이다. 이 때 연결은 완전히 종료된다. 클라이언트 소켓은 TIME_WAIT 상태로 들어간다.
7. 프로세스 종료의 또 다른 부분은 서버 자식이 종료할 때 SIGCHLD신호를 부모에게 보내는 것이다. 이 예에서 그것이 발생하지만 프로그램에서는 그 신호를 포착하지 않고 신호의 기본 동작도 무시한다. 자식은 좀비 상태로 들어간다. 이는 ps 명령을 이용하여 확인할 수 있다.

```
linux % ps -t pts/6 -o pid,ppid,tty,stat,args,wchan
  PID  PPID TT       STAT COMMAND          WCHAN
22038 22036 pts/6    S    -bash            read_chan
17870 22038 pts/6    S    ./tcpserv01       wait_for_connect
19315 17870 pts/6    Z    [tcpserv01 <defu do_exit
```

좀비 프로세스들을 제거할 필요가 있으며, 그러기 위해서는 유닉스의 신호들을 다룰 필요가 있다. 다음 절에서 신호 처리의 overview를 살펴본다.

5.4 시그널 처리함수

A. 시그널 처리

신호는 프로세스에게 어떤 사건이 발생했음을 알리는 것이다. 신호는 때때로 software interruption이라고 부른다. 신호는 대부분 비동기로 발생한다. 이는 프로세스가 신호가 발생하는 정확한 순간을 미리 알지 못한다는 것을 의미한다.

- 하나의 프로세스가 다른 프로세스에게 (또는 그 자신에게)
- 커널이 프로세스에게

앞 절의 마지막 부분에서 설명한 SIGCHLD 신호는 프로세스가 종료될 때마다 커널이 종료하는 프로세스에 보내지는 것이다.

모든 신호는 신호와 연관된 action이라고 불리는 disposition을 가진다. Sigaction() 함수를 호출함으로서 신호의 disposition을 세팅하고 disposition에 대한 세 가지 선택을 가진다.

① 특정 신호가 발생할 때마다 호출할 함수를 제공한다. 이 함수를 signal handler라고 부르며, 함수 호출을 신호 포착이라고 부른다. SIGKILL과 SIGSTOP은 포착할 수 없는 두 개의 신호이다. 신호 번호인 하나의 정수 인수로써 쌍수를 호출한다. 그리고 함수는 아무 것도 돌려주지 않는다. 이 함수 원형은 따라서,

void handler(int signo);

이다. 대부분의 신호에 대해서, sigaction을 호출하여 신호가 발생할 때 호출할 함수를 지정하는 것은 신호를 포착하는 데 필요한 모든 것이다. 그러나 후에 SIGIO, SIGPOLL 같은 몇몇 신호는 그 신호를 포착할 프로세스의 일부분에 추가적인 조치를 필요로 한다.

② disposition을 SIG_IGN으로 설정함으로써 신호를 무시할 수 있다. 신호 SIGKILL과 SIGSTOP은 무시할 수 없다.

③ disposition을 SIG_DFL로 설정함으로써 신호의 기본 disposition을 설정할 수 있다. 기본은 보통 신호를 받으면 프로세스를 종료시키는 것이며, 어떤 신호에서는 프로세스의 메모리 내용을 그대로 현재의 working directory에 옮겨놓기도 한다. 기본 disposition을 무시하라는 것인 몇몇 신호도 있다. SIGCHLD 신호가 이에 해당한다.

Signal() Function

POSIX에서 신호의 disposition을 설정하는 방법은 sigaction 함수를 호출하는 것이다. 이 함수의 인수는 반드시 할당하고 채워야 할 구조이기에 복잡해진다. 신호의 disposition을 설정하는 가장 쉬운 방법은 signal 함수를 호출하는 것이다. 첫 번째 인수는 신호의 이름이고 두 번째 인수는 함수를 가리키는 pointer이거나 상수 SIG_IGN과 SIG_DFL 중 하나이다.

그러나 signal은 POSIX 이전의 함수이고 구현에 따라 의미가 달라진다. 반면에 POSIX에서는 sigaction을 호출할 때 명확하게 그 의미를 적시할 수 있다. 해결책으로, POSIX의 sigaction을 호출하는 signal이라는 이름을 가진 고유의 함수를 정의한다. 이 함수를 다음 예제에서 보여준다.

```
<signal.c>
--------------------------------------------------------------------------
1 #include    "lnp.h"
2 Sigfunc *
3 signal (int signo, Sigfunc *func)
4 {
5     struct sigaction act, oact;

6     act.sa_handler = func;
7     sigemptyset (&act.sa_mask);
8     act.sa_flags = 0;
9     if (signo == SIGALRM) {
10 #ifdef  SA_INTERRUPT
11         act.sa_flags |= SA_INTERRUPT;     /* SunOS 4.x */
12 #endif
13     } else {
14 #ifdef  SA_RESTART
15         act.sa_flags |= SA_RESTART; /* SVR4, 4.4BSD */
16 #endif
17     }
18     if (sigaction (signo, &act, &oact) < 0)
19         return (SIG_ERR);
20     return (oact.sa_handler);
21 }
--------------------------------------------------------------------------
```

Simplify function prototype using typedef

2-3

signal에 대한 정상적인 함수 원형은 중첩된 괄호에 의해 매우 복잡하다.

```
void (*signal (int signo, void (*func) (int))) (int);
```

이를 단순화하기 위하여 lnp.h 헤더에 다음과 같이 Sigfunc형을 정의한다.

```
typedef   void   Sigfunc(int);
```

이것은 signal handler는 정수 인수를 가지며, 아무 것도 반환하지 않는 함수라고 말한다(void). 함수 원형은 다음과 같다.

```
Sigfunc *signal (int signo, Sigfunc *func);
```

신호를 처리하는 함수를 가리키는 pointer는 함수의 두 번째 인수이면서, 함수로부터 반환 받는 값이다.

Set handler

6 sigaction 구조의 sa_handler 원소는 **func** 인수로 설정한다.

Set signal mask for handler

7

POSIX에서 signal handler가 호출될 때 블록될 신호들의 집합을 지정할 수 있다. 블록된 신호는 프로세스에 전달될 수 없다. Sa_mask를 공집합으로 지정하면 signal handler가 수행되고 있는 동안에는 어떤 추가의 신호도 블록되지 않음을 의미한다. POSIX에서는 포착할 신호는 handler가 수행되는 동안에 항상 블록된다.

Set SA_RESTART flag

8-17

SA_RESTART는 optional flag이다. 이것이 켜져 있으면, 이 신호에 의해 interrupted system 호출은 자동으로 커널에 의해 다시 시작될 것이다.(다음 장에서 interrupted system 호출에 대해서 좀 더 설명할 것이다.) 만약 포착된 신호가 SIGALRM이 아니라면 SA_RESTART flag(정의되어 있으면)를 지정한다. (SIGALRM을 특수한 경우로 여기는 이유는 Section 14.2에서 보는 것과 같이 이 신호를 발생시키는 목적은 일반적으로 입 출력 동작에 시간 만료를 설정하는 것인데, 이 경우에 신호가 블록된 시스템 호출을 일시 중지하도록 하는 것이다.) SunOS 4.x같은 오래된 시스템에서는 interrupted system 호출을 자동으로 시작하는 것이 기본이다. 이 flag의 complement를 SA_INTERRUPT로 정의한다. 만약 이 flag가 정의되고 포착할 신호가 SIGALRM이면 이 flag를 켠다.

Call sigaction

18-20

sigaction을 호출하고 signal 함수의 반환 값으로 신호에 대한 이전 action을 돌려준다. 이 책에서는 signal 함수를 사용한다.

POSIX Signal Semantics

다음은 POSIX를 따르는 시스템에서 신호 처리에 대한 것들의 요약이다.

- 한번 signal handler를 설치하면 설치된 채로 남는다.
- signal handler가 실행되는 동안 전달될 신호는 블록된다. 더욱이 signal handler가 설치 될 때 sigaction으로 전달된 sa_mask 신호 집단에서 지정한 어떤 추가 신호도 또한 블록킹된다. 예제에서는 sa_mask를 공집합으로 설정하는데, 이는 포착할 신호 이외의 어떤 신호도 블록킹하지 않는다는 뜻이다.
- 만약 신호가 블록되어 있는 동안에도 한 번 이상 발생하면 블록 상태에서 해제된 후에는 한 번만 전달된다. 즉 기본으로 유닉스 신호는 queue에 쌓이지 않는다. 다음 절에서 이 예제를 볼 것이다.

B. SIGCHLD 시그널 처리

좀비 상태의 목적은 나중에 부모가 가져갈 자식에 대한 정보를 유지하는 것이다. 이 정보는 자식의 프로세스 ID와 종료 상태 및 자식의 자원 사용률(CPU 시간, 메모리 등)을 포함하고 있다. 만약 프로세스가 종료하면서 좀비 상태에 있는 자식을 가지고 있으면, 모든 좀비 자식의 부모 프로세스 ID는 1(init 프로세스)로 바뀌며, 그 프로세스는 자식을 넘겨 받아 청소한다. 즉 좀비를 제거한다(즉, init는 그들을 wait한다) 어떤 유닉스 시스템에서는 좀비 프로세스에 대한 COMMAND행을 <defunct>로 보여준다.

Handling Zombies

좀비를 내버려 두어서는 안 된다. 좀비는 커널의 공간을 차지하므로, 궁극에는 프로세스를 더 이상 만들 수가 없게 한다. 자식을 fork할 때마다 자식이 좀비가 되는 것을 방지하기 위해 반드시 자식을 wait해야 한다. 이렇게 하기 위하여 SIGCHLD를 포착하는 signal handler를 설정하여 그 signal handler 안에서 wait를 호출한다.

예제에서 listen호출 다음에 다음 함수 호출을 추가함으로써 signal handler를 설정한다.

```
Signal (SIGCHLD, sig_chld);
```

이는 반드시 첫 자식을 fork하기 전에 행해야 하고 한 번이면 충분하다. 그리고 나서 signal handler 함수를 정의한다.

```
<sigchldwait.c>
--------------------------------------------------------------------------
 1 #include     "unp.h"

 2 void
 3 sig_chld(int signo)
 4 {
 5    pid_t   pid;
 6    int     stat;

 7    pid = wait(&stat);
 8    printf("child %d terminated\", pid);
 9    return;
10 }
-------------------------------------------------------------------------
```

만약 이 프로그램을 Solaris 9에서 컴파일하고 시스템 라이브러리의 signal 함수를 사용하면 다음 결과를 얻는다.

```
solaris % tcpserv02 &                  start server in background
[2] 16939
solaris % tcpcli01 127.0.0.1           then start client in foreground
hi there                               we type this
hi there                               and this is echoed
^D                                     we type our EOF character
child 16942 terminated                 output by printf in signal handler
accept error: Interrupted system call  main function aborts
```

상기 실험의 절차는 다음과 같다.

1. EOF문자를 쳐 넣어 클라이언트를 끝나게 한다. 클라이언트 TCP는 서버에게 FIN을 보내고 서버는 ACK로 응답한다.
2. FIN을 수신하면 자식의 미해결의 readline에 EOF를 전달한다. 자식은 종료한다.
3. 부모는 SIGCHLD 신호가 배달될 때 accept 호출에서 블록되어 있다. sig_chld함수(signal handler)는 실행되어, wait는 자식의 PID와 종료 상태를 가져오고 printf가 호출된다.
4. 부모가 disposition(accept 호출)에서 블록되어 있는 동안에, 부모는 신호를 포착하므로 커널은 accept가 EINTR(interrupted system 호출) 오류를 돌려주게 된다. 부모는 이 오류를 처리하지 않아 중단하게 된다.

이 예제의 목적은 signal을 처리하는 네트워크 프로그램을 작성할 때, 일시 정지된 시스템 호출을 반드시 인식하고 처리해야만 한다는 것을 보여주는 데 있다. Solaris 9에서 실행하는 이 특정 예에서 표준 C 라이브러리가 제공하는 signal 함수는 interrupted system 호출을 커널이 자동으로 재시작하도록 하지 않는다. 즉 예제에서 지정한 SA_RESTART flag는 시스템 라이브러리의 signal 함수가 지정하지 않는다. 어떤 시스템에서는 자동으로 interrupted system 호출을 재시동한다. 만약 4.4BSD 환경에서 라이브러리에 있는 signal 함수를 사용하여 같은 예제를 실행하면 커널은 interrupted system 호출을 재시작하고 accept는 오류를 반환하지 않는다.

signal handler에서 return 문을 사용하지만 함수의 끝에 이르러서는 void를 돌려주는 함수와 같은 일을 한다. 코드를 읽을 때 불필요한 return statement는 시스템 호출을 일시 중지할 수도 있음을 경고하는 것이다.

Handling Interrupted System Calls

accept를 기술하는 데 "disposition"이란 용어를 사용했고 이 용어는 영원히 블록될 수 있는 시스템 호출을 의미한다. 즉, return 될 수 없는 시스템 호출을 이야기 한다. 대부분의 네트워크 함수는 이 범주에 속한다. 예를 들어, 만약 클라이언트가 연결을 요청하지 않는다면 서버의 accept호출이 돌아온다는 보장이 없다. 비슷한 경우로 만일 클라이언트가 서버가 echo할 행을 보내지 않으면 read(readline)가 반환되지 않을 것이다. disposition의 다른 예는 파이프와 단말기에서의 입출력이다. 주목해야 할 예외는 caller에게 통산 돌아가는 디스크 입출력이다.

여기에서 적용되는 기본적인 규칙은 프로세스가 disposition에서 블록되어 신호를 포착하고 signal handler에서 돌아올 때 시스템 호출은 EINTR 오류를 돌려줄 수 있다는 것이다. 어떤 커널은 자동으로 interrupted system 호출을 다시 시작한다. 이식성을 위해 신호를 포착하는 프로그램을 작성할 때(대부분 다중 접속 서버는 SIGCHLD를 포착한다), 반드시 disposition이 EINTR을 돌려주도록 해야 한다. 이식성 문제는 POSIX SA_RESTART flag를 지원하는 것은 선택 옵션이라는 사실과 초기에 사용한 'can'과 'some'같은 용어에 기인한다. 비록 구현에서 SA_RESTART flag를 지원한다고 할지라도 모든 interrupted system 호출이 자동으로 다시 시작하는 것은 아니다. 예를 들어, 대부분의 Berkeley-derived implementations에서 자동으로 select를 재시작하지 않으며 이들 중 몇몇에서는 accept 또는 recvfrom을 결코 재시작하지 않는다.

일시 중지된 accept를 처리하기 위하여 accept 호출, 즉 for 반복문의 시작 부분을 아래와 같이 변경한다.

```
for ( ; ; ) {
    clilen = sizeof (cliaddr);
    if ( (connfd = accept (listenfd, (SA *) &cliaddr, &clilen)) < 0) {
        if (errno == EINTR)
            continue;        /* back to for () */
        else
            err_sys ("accept error");
   }
```

wrapper 함수 Accept가 아닌 accept를 호출함을 알아야 한다. 이는 함수의 실패를 직접 처리해야 하기 때문이다. 이 부분의 코드에서 하고자 하는 일은 interrupted system 호출을 직접 다시 시작하는 것이다. 이는 read, write, select, open 함수와 함께 accept 함수에 있어서도 괜찮다. 그러나 다시 시작할 수 없는 함수가 하나 있는데 그것은 connect이다. 만약 이 함수가 EINTR을 돌려준다면 다시 이 함수를 호출할 수 없다. 왜냐하면 다시 호출할 경우 즉각적으로 오류가 발생하기 때문이다. 포착 신호가 자동으로 다시 시작하지 않는 connect를 일시 중지시키면 연결이 완성되기를 기다리는 select를 호출해야 한다.

C. wait 및 waitpid 함수

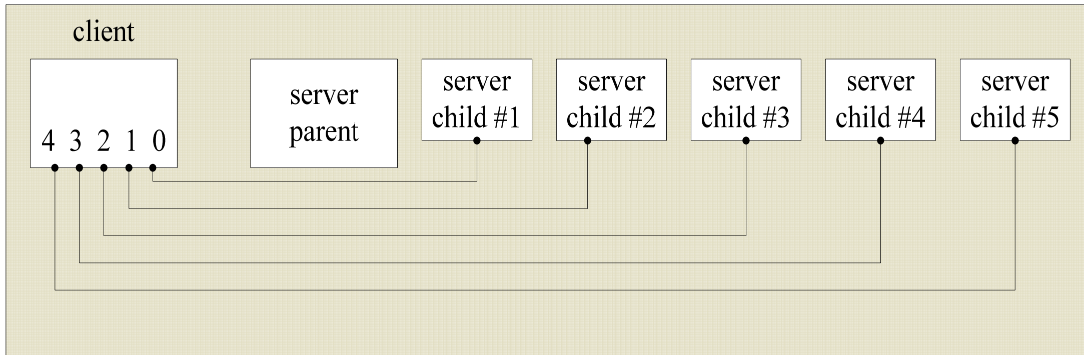
종료된 자식을 처리하기 위해 wait함수를 호출하였다.

```
#include <sys/wait.h>

pid_t wait (int *statloc);
pid_t waitpid (pid_t pid, int *statloc, int options);
Both return: process ID if OK, 0 or-1 on error
```

wait와 waitpid는 두 개의 값을 돌려준다. 이들은 함수의 return 값으로써 종료된 자식의 프로세스 ID와 statloc pointer가 가리키는 자식의 종료 상태(상수)이다. 종료 상태를 검사하여 자식이 정상적으로 종료되었는지, 신호에 의해서 끝났는지 혹은 오직 작업 제어에 의해서 정지되었는지를 알려주는 데 사용하는 세 개의 매크로가 있다. 다른 매크로를 이용하여 자식의 종료 상태 또는 자식을 끝나게 한 신호의 값 또는 자식을 정지 시킨 작업 제어 신호를 파악할 수 있다. 이런 목적으로 WIFEXITED와 WEXITSTATUS 매크로를 사용한다.

Wait를 호출한 프로세스의 하나 이상의 수행 중인 자식은 있지만 종료된 자식이 없으면 wait에서 어느 자식 프로세스가 처음으로 종료할 때까지 프로세스는 블록된다. waitpid로는 어느 프로세스를 기다릴 것인지와 블록될 것인지를 조절할 수 있다. 먼저 pid 인수로 기다리려고 하는 프로세스 ID를 지정한다. -1은 어느 자식이나 처음 종료되기를 기다리라는 것이다. (프로세스 집단 ID를 다루는 다른 선택 사항도 있지만 이 책에서는 필요하지 않다.) options 인수는 추가 선택 사항들을 지정한다. 가장 일반적인 선택 사항은 WNOHANG이다 이것은 커널에게 종료될 자식 프로세스가 없으면 블록하지 말도록 하고 아직 작업 중인 자식 프로세스가 있으면 블록하도록 한다.

Wait와 waitpid 함수가 종료된 자식들을 없앨 때의 차이를 예시한다. 이를 위해 TCP클라이언트를 다음 예제와 같이 바꾼다. 클라이언트는 서버와 다섯 개의 연결을 설정하고 str_cli를 호출하는데 첫 째(sockfd[0])만을 사용한다. 다중 연결의 목적은 그림 5.2와 같이 다중 접속 서버가 다중 자식 프로세스를 만들게 하는 것이다.



<그림 5.2> 서버와 5개의 연결을 갖는 클라이언트 예제

<tcpcli04.c>

```
------------------------------------------------------------------------------
 1 #include     "unp.h"

 2 int
 3 main (int argc, char **argv)
 4 {
 5      int     i, sockfd[5];
 6      struct sockaddr_in servaddr;

 7      if (argc != 2)
 8          err_quit ("usage: tcpcli <IPaddress>";

 9      for (i = 0; i < 5; i++) {
10          sockfd[i] = Socket (AF_INET, SOCK_STREAM, 0);

11          bzero (&servaddr, sizeof (servaddr));
12          servaddr.sin_family = AF_INET;
13          servaddr.sin_port = htons (SERV_PORT);
14          Inet_pton (AF_INET, argv[1], &servaddr.sin_addr);

15          Connect (sockfd[i], (SA *) &servaddr, sizeof (servaddr));
16     }

17     str_cli (stdin, sockfd[0]);  /* do it all */

18     exit(0);
19 }
------------------------------------------------------------------------------
```

다음은 tcpcli04.c 예제의 실행 종료 후 server의 실행화면이다.

```
[LeeDonghwa@localhost tcpcliserv]$ ./tcpserv04
child 29046 terminated
child 29047 terminated
child 29048 terminated
child 29049 terminated
child 29050 terminated
```

클라이언트가 종료하면 커널은 모든 open descriptors를 자동으로 닫고(close를 호출하지 않고 exit만을 호출한다.), 거의 동시에 다섯 개의 연결을 모두 종료한다. 그래서 각 연결에 하나씩 다섯 개의 FIN을 보내고 이번에는 다섯 개의 서버 자식을 거의 동시에 모두 종료한다. 이것은 다음 그림에서와 같이 거의 동시에 다섯 개의 SIGCHLD 신호를 부모에게 전달한다.

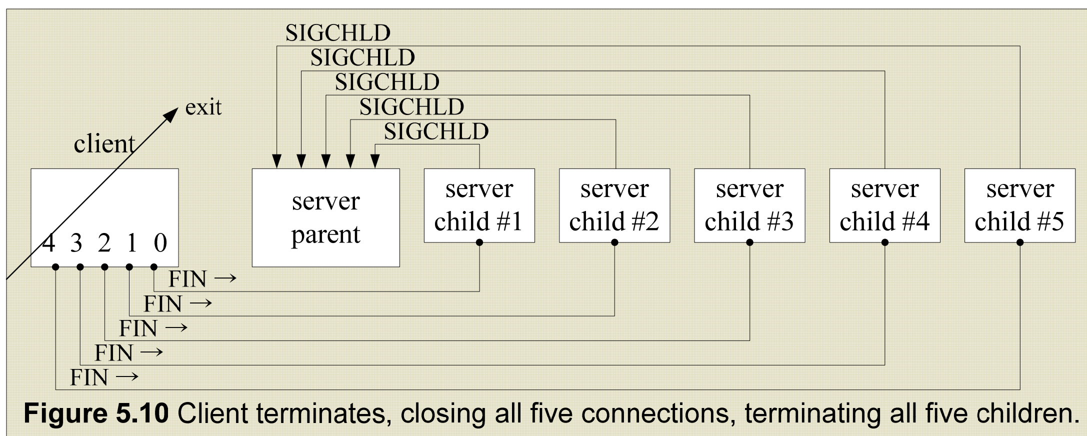


Figure 5.10 Client terminates, closing all five connections, terminating all five children.

<그림 5.3> 5개의 child 프로세스를 종료하는 예제

여러 번 발생된 같은 신호를 전달하는 것은 다음에서 보게 될 문제를 야기시킨다. 먼저 background에서 서버를 실행한 후 새로운 클라이언트를 실행한다. 예제 코드에서 SIGCHLD에 대한 signal handler를 호출하도록 수정한다.

```
linux % tcpserv03 &
[1] 20419
linux % tcpcli04 127.0.0.1
hello                              we type this
hello                              and it is echoed
^D                                 we then type our EOF character
child 20426 terminated             output by server
```

첫 번째로 주목할 것은 다섯 개의 모든 자식이 종료했을 것으로 기대하지만, 단지 출력은 하나의 printf라는 것이다. ps를 실행해 보면 나머지 네 개의 자식이 좀비로서 남아 있음을 볼 수 있다.

```
PID TTY         TIME CMD
20419 pts/6    00:00:00 tcpserv03
20421 pts/6    00:00:00 tcpserv03 <defunct>
20422 pts/6    00:00:00 tcpserv03 <defunct>
20423 pts/6    00:00:00 tcpserv03 <defunct>
```

signal handler를 만들고 이 handler에서 wait를 호출한다 해도 좀비를 방지하기에는 충분하지 않다. 문제는 다섯 개의 모든 신호가 signal handler가 실행되기 전에 발생하고, 유닉스 신호들은 보통 queue에 넣지 않기 때문에 signal handler는 한 번만 실행하게 된다. 더구나 이 문제는 결정적인 것이 아니다. 이 예에서 같은 호스트에서 클라이언트와 서버를 실행하므로 signal handler가 한 번만 실행되어 네 개의 좀비를 남긴다. 그러나 각기 다른 호스트에서 클라이언트와 서버를 실행하면 signal handler는 보통 두 번 실행된다. 한 번은 첫 번째 신호에 의해서이고, signal handler가 실행하는 도안 네 개의 신호가 발생하여 한 번 더 signal handler를 호출한다. 따라서 세 개의 좀비가 남게 된다. 그러나 때때로, 서버 호스트에 도착하는 FIN의 시간에 따라서, signal handler는 3회 또는 심지어 4회 실행되기도 한다.

올바른 해결 방안은 wait 대신에 waitpid를 호출하는 것이다. Figure 5.11에서 SIGCHLD를 올바르게 다루는 새로운 sig_칭 함수를 보여준다. 종료한 자식 중 어느 하나의 상태를 가져오는 waitpid를 반복문 안에서 호출하기 때문에 이것은 제대로 동작한다. 여기서 WNOHANG 선택 사항을 지정해야만 하는데, 이로써 종료되지 않은 자식이 실행 중이면 waitpid가 블록되지 않도록 한다. 이전 예제에서는 반복문 안에서 wait를 호출할 수가 없다. 그 이유는 아직 종료되지 않은 자식이 있을 때 wait가 블록되지 않도록 하는 방법은 없기 때문이다.

아래 예제에서 서버 코드의 최종판을 보여준다. 여기서 accept에서 돌아오는 EINTR을 올바르게 처리하고 모든 종료된 자식들에 대하여 waitpid를 호출하는 signal handler를 설정한다.

<sigchldwaitpid.c>

```
---------------------------------------------------------------------------
 1 #include    "unp.h"

 2 void
 3 sig_chld(int signo)
 4 {
 5    pid_t   pid;
 6    int     stat;

 7    while ( (pid = waitpid(-1, &stat, WNOHANG)) > 0)
 8       printf("child %d terminated\n", pid);
 9    return;
10 }
---------------------------------------------------------------------------
```

<tcpserv04.c>

```
-----------------------------------------------------------------------
1 #include    "unp.h"
2 int
3 main(int argc, char **argv)
4 {
5    int    listenfd, connfd;
6    pid_t   childpid;
7    socklen_t clilen;
8    struct sockaddr_in cliaddr, servaddr;
9    void   sig_chld(int);

10    listenfd = Socket (AF_INET, SOCK_STREAM, 0);

11    bzero (&servaddr, sizeof(servaddr));
12    servaddr.sin_family = AF_INET;
13    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
14    servaddr.sin_port = htons(SERV_PORT);

15    Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

16    Listen(listenfd, LISTENQ);

17    Signal (SIGCHLD, sig_chld); /* must call waitpid() */

18    for ( ; ; ) {
19       clilen = sizeof(cliaddr);
20        if ( (connfd = accept (listenfd, (SA *) &cliaddr, &clilen)) < 0)
{
21          if (errno == EINTR)
22             continue;        /* back to for() */
23          else
24             err_sys("accept error");
25       }

26       if ( (childpid = Fork()) == 0) { /* child process */
27          Close(listenfd);    /* close listening socket */
28          str_echo(connfd);   /* process the request */
29          exit(0);
30       }
31       Close (connfd);        /* parent closes connected socket */
32    }
33 }
-----------------------------------------------------------------------
```

지금까지 시그널처리 과정에서 다음 사항을 명심해야한다.

1) 자식 프로세스를 fork할 때 SIGCHLD 신호를 포착해야 한다.

2) 신호를 포착할 때 interrupted system 호출을 처리해야 한다.

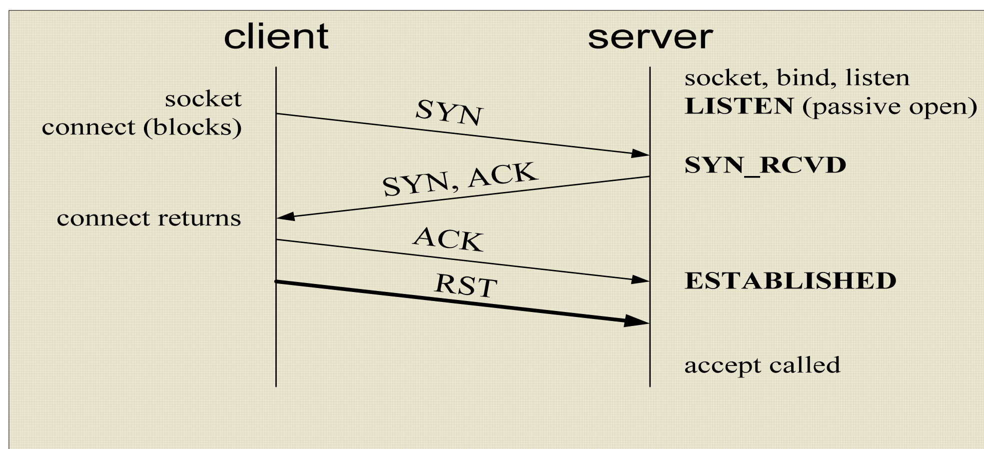
3) SIGCHLD handler는 좀비가 남지 않도록 waitpid를 사용하여 올바르게 해결해야 한다.

5.5 비정상적인 연결 종료

A. Connection Abort before Accept Returns

앞 절의 interrupted system 호출의 예와 비슷한 또 다른 상황이 있는데, 이는 accept함수가 치명적이 아닌 오류를 돌려주도록 한다. 이 경우 다시 accept를 호출하면 된다.

다음 그림에서 보여주는 패킷의 순서는 busy server(예: Web servers)에서 볼 수 있다.



<그림 5.4> accept 호출되기 전에 연결해제

three-way handshake가 완료되어 연결이 이루어진 후에, 클라이언트 TCP가 RST(reset)을 보낸다. 서버쪽에서는 RST가 도착할 때 accept호출을 서버 프로세스가 하기를 기다리면서, TCP는 연결을 queue에서 기다리게 한다. 잠시 후에 서버 프로세스는 accept를 호출한다.

이 시나리오를 모의 시험하는 쉬운 방법은 서버를 시작하여 socket, bind, listen을 차례로 호출한 후, accept 호출 전에 짧은 시간 동안 잠들게 한다. 서버가 잠자는 동안에 클라이언트를 시작하고 socket과 connect를 차례로 호출한다. connect에서 돌아오자 마자 RST를 발생시키는 SO_LINGER 소켓 선택 사항을 지정하고 종료한다.

취소된 연결에 무슨 일이 일어나는지는 구현에 따라 다르다. Berkeley-derived implementations에서는 취소된 연결을 완전히 커널 내에서 다루고 서버는 전혀 그것을 알지 못한다. 그러나 대부분의 SVR4 구현에서는 accept가 돌려주는 값으로 오류를 프로세스에게 돌려주며, 어떤 오류인지는 구현에 따라 다르다.

B. 서버 프로세스의 종료

클라이언트/서버를 시작한 후에 서버 자식 프로세스를 종료한다. 이것은 서버 프로세스의 갑작스러운 종료를 시험하는 것인데, 클라이언트에 무슨 일이 일어나는지를 볼 수 있다. 다음과 같은 단계에 따라 실험을 수행한다.

서버와 클라이언트를 서로 다른 호스트에서 시작하여, 이 때 모든 것이 정상임을 검증하기 위해 클라이언트에 한 줄을 입력해 본다. 서버 자식은 그 입력을 그대로 echo 해준다.

서버의 자식 프로세스 ID를 찾아 kill한다. 프로세스 종료의 한 부분으로 자식 내의 모든 open descriptors는 닫힌다. 그 결과 FIN이 클라이언트로 보내지며, 클라이언트 TCP는 ACK로써 응답한다. 이것은 TCP 연결 종료의 처음 절반에 해당한다. SIGCHLD 신호가 서버 부모에게 보내어져 적절히 처리된다.

클라이언트에서는 아무 일도 일어나지 않는다. 클라이언트 TCP는 서버 TCP로부터 FIN을 받고 ACK로 응답한다. 그러나 문제는 클라이언트 프로세스가 단말기로부터 문자열을 기다리는 fgets 호출에서 블록되어 있다는 것이다.

이 때 클라이언트의 다른 창에서 netstat를 실행하면 클라이언트 소켓의 상태를 볼 수 있다.

```
linux % netstat -a | grep 9877
tcp       0      0 *:9877               *:*                 LISTEN
tcp       0      0 localhost:9877       localhost:43604     FIN_WAIT2
tcp       1      0 localhost:43604      localhost:9877      CLOSE_WAIT
```

TCP 연결 종료 과정의 절반이 이루어졌음을 알 수 있다. 클라이언트에 한 줄을 입력해 본다. 다음은 클라이언트에서 일어나는 것이다.

```
linux % tcpcli01 127.0.0.1      start client
hello                           the first line that we type
hello                           is echoed correctly here we kill the
                                server child on the server host
```

```
linux % tcpcli01 127.0.0.1      start client
another line                    we then type a second line to the client
str_cli : server terminated
prematurely
```

'another line'을 입력할 때, str_cli는 writen을 호출하고 클라이언트 TCP는 서버에게 데이터를 보낸다. 이것이 TCP에서 허락되는 이유는 클라이언트 TCP의 FIN 수신은 서버 프로세스가 연결의 서버측 끝을 닫고 더 이상의 데이터를 보내지 않을 것임을 뜻하기 때문이다. FIN 수신으로 클라이언트 TCP가 서버 프로세스가 종료되었음을 알려주지는 못한다. 서버 TCP가 클라이언트로부터 데이터를 받으면 RST로 응답하는데, 그 이유는 그 소켓을 열었던 프로세스가 종료하였기 때문이다. tcpdump로 패킷을 살펴보면 RST가 보내졌음을 알수 있다.

그러나 클라이언트 프로세스는 RST를 볼 수 없을 것이다. 왜냐하면 클라이언트가 writen을 호출한 다음에 곧바로 readline을 호출하는데, readline은 단계 2에서 받은 FIN 때문에 곧바로 0(EOF)을 돌려주기 때문이다. 이 때 클라이언트는 'EOF'를 받을 것이라고 기대하지 않았으므로 "server terminated prematurely."라는 오류 메시지와 함께 종료한다. 클라이언트가 종료할 때 모든 open descriptors를 닫는다.

여기서 설명한 것은 또한 예제가 일어나는 시간에 따라 다르다. 방금 설명한 것처럼 클라이언트와 서버를 다른 호스트에서 실행할 때, 클라이언트가 데이터('another line')를 서버로 보내고, 서버가 보낸 RST를 클라이언트가 받기까지는 수 ms가 걸린다. 이것이 클라이언트의 readline 호출이 0을 돌려주는 이유인데, 일찍 수신한 FIN은 이때 읽혀질 준비가 되어 있기 때문이다. 그러나 클라이언트와 서버를 같은 호스트에서 실행하거나 readline을 호출하기 전에 클라이언트를 잠시 정지시키면, RST가 일찍 받은 FIN보다 우선하게 된다. 그리하여 readline은 errno가 ECONNRESET(상대에 의한 연결 취소)인 오류를 돌려준다.

이 예제에서 문제는 FIN이 소켓에 도착할 때 클라이언트가 fgets 호출에서 블록되어 있다는 것이다. 클라이언트는 실제로 소켓과 사용자 입력의 두 descriptors를 가지고 작동한다. 두 descriptor 중 하나의 입력에서 블록되는 대신에 어느 쪽 입력에서나 블록될 수 있어야 한다. 이것이 바로 6장에서 설명할 select와 poll 함수의 목적이다. 6장에서 str_cli함수를 다시 작성하는데, 서버 자식을 kill하자마자 클라이언트는 FIN 수신을 통지 받게 된다.

5.6 SIGPIPE 시그널

클라이언트가 readline에서 오는 오류를 무시하고 서버에게 데이터를 쓰면 어떻게 될까? 예를 들어, 클라이언트가 서버에서 echo되는 것을 읽기 전에 두 번 서버에 데이터를 전송하고 첫 번째 전송이 RST를 유발하면 이런 일이 일어난다.

이에 적용되는 규칙은 프로세스가 RST를 받은 소켓에 데이터를 쓰면, 그 프로세스에게 SIGPIPE 신호를 보내는 것이다. 이 신호에 있어 기본 동작은 프로세스를 종료시키는 것이므로, 프로세스가 원하지 않는 종료를 피하기 위해서는 이 신호를 포착해야 한다.

만약 프로세스가 그 신호를 포착하여 signal handler로부터 돌아오거나 또는 신호를 무시하면, write()는 EPIPE를 반환한다.

SIGPIPE에 대해 알아보기 위하여 클라이언트를 다음과 같이 수정한다.

```
<str_cli11.c>
-----------------------------------------------------------------------------
 1 #include    "unp.h"

 2 void
 3 str_cli(FILE *fp, int sockfd)
 4 {
 5    char   sendline [MAXLINE], recvline [MAXLINE];

 6    while (Fgets(sendline, MAXLINE, fp) != NULL) {

 7        Writen(sockfd, sendline, 1);
 8        sleep(1);
 9        Writen(sockfd, sendline + 1, strlen(sendline) - 1);

10        if (Readline(sockfd, recvline, MAXLINE) == 0)
11           err_quit("str_cli: server terminated prematurely");

12        Fputs(recvline, stdout);
13    }
14 }
-----------------------------------------------------------------------------
```

7-9

바뀐 것은 writen을 두 번 호출하는 것이다. 먼저 데이터의 첫 바이트를 소켓에 쓰고, 1초 동안 기다린 후 나머지를 수행한다. 목적은 처음 writen으로 RST를 유발한 다음 두 번째 writen으로 SIGPIPE신호를 발생시키는 것이다.

만약 BSD/OS 호스트에서 클라이언트를 실행하면, 결과는 다음과 같다.

```
linux % tcpclill 127.0.0.1
hi there              we type this line
hi there              this is echoed by the server
                      here we kill the server child
bye                   then we type this line
Broken pipe           this is printed by the shell
```

클라이언트를 시작하여 한 줄을 입력하면 올바르게 echo되는 것을 보게 되며, 이 때 서버 호스트의 서버 자식은 종료한다. 다음 줄('bye')을 입력하고 shell은 SIGPIPE signal과 함께 프로세스가 종료되었음을 알린다.

권장하는 SIGPIPE 처리 방법은 이 신호가 발생할 때 응용에 따라 다르다. 특별히 해야 할 것이 없다면 다음의 출력 동작이 EPIPE 오류를 포착하여 종료한다고 가정하면서 SIG_IGN으로 신호 disposition을 설정하는 것이 쉽다. 신호가 발생할 때 특별한 동작이 필요하면 신호를 포착해야 하고 signal handler에서 필요한 동작을 행할 수 있다. 하지만 여러 소켓을 사용할 때는 신호가 오더라도 어느 소켓에서 오류가 발생했는지 알 수 없다. 그러므로 어느 write가 오류를 일으켰는지 알려면, 신호를 무시하거나 또는 signal handler에서 돌아온 후에 write의 EPIPE를 처리해야만 한다.

5.7 ABNORMAL SERVER TERMINATION

A. 서버 Host의 Crashing

다음 시나리오로 서버 호스트가 갑자기 정지할 때는 무슨 일이 발생하는지를 알아본다. 이를 위해 클라이언트와 서버를 각기 다른 호스트에서 실행한다. 서버와 클라이언트를 시작하여 연결되었는지를 알아보도록 클라이언트에 한 줄을 입력한다. 그런 다음 서버를 네트워크에서 분리시키고 클라이언트에 두 번째 입력을 한다. 이 시나리오는 클라이언트가 데이터를 보낼 때 도착 불가능한 서버 호스트 경우도 포함한다. (즉, 연결 설정 후 중간 라우터 고장)

다음과 같은 단계를 따른다.

1. 서버 호스트가 멈추면 설정된 네트워크 연결에 아무 것도 보내지 않는다. 즉, 운영자가 중단시킨 것이 아닌 호스트 장애로 가정한다.

2. 클라이언트에 한 줄을 입력한다. 입력된 데이터는 writen()으로 쓰고, 클라이언트 TCP가 데이터 세그먼트로 보낸다. 클라이언트는 readline 호출에서 echo 응답을 기다리며 블록된다.

3. tcpdump를 이용하여 네트워크를 관찰하면, 클라이언트 TCP는 서버로부터 ACK를 받기 위해 데이터 세그먼트를 계속해서 재전송한다. 데이터 세그먼트를 12회 재전송하면서 포기할 때까지 약 9분 동안 기다린다. 클라이언트 TCP가 데이터 전송을 포기하면(서버 호스트가 이 시간 동안 재시동되지 않았다고 가정하거나 또는 서버 호스트에서 장애가 생기지 않았지만 네트워크에서 도달 불가능하면 호스트가 도달 불가능하다고 가정하면서) 클라이언트 프로세스에게 오류를 돌려준다. 클라이언트는 readline 호출에서 블록되었기 때문에 오류를 반환한다. 서버가 중단되어 클라이언트의 데이터 세그먼트에 대해 응답이 없을 때의 오류는 ETIMEDOUT이다. 그러나 만약 중간 라우터가 서버에 도달할 수 없다고 결정하고 목적지 도달 불가능이라는 ICMP를 보내면, 오류는 EHOSTUNREACH 또는 ENETUNREACH이다.

결국 클라이언트는 상대가 고장났거나 도달할 수 없음을 알게 되겠지만, 9분 동안 기다리는 것보다 빨리 발견하기를 원한다. 해결책은 readline 호출에 시간 만료 timer를 설정하는 것이다.

지금까지 논의한 바로는 호스트에 데이터를 보낼 때만 서버 호스트에 장애가 발생했음을 알 수 있다. 데이터를 보내지 않고서 서버 호스트가 정지되었음을 발견하려면 다른 기법이 필요하다.

B. 서버 Host의 Crashing과 Rebooting

이번 시나리오에서는 클라이언트와 서버 간의 연결이 이루어진 후에 서버 호스트가 갑자기 정지하여 다시 시작한다. 앞 절에서 서버에 데이터를 보낼 때 서버는 계속 정지되어 있었다. 여기서는 서버에게 데이터를 보내기 전에 서버 호스트를 재시동한다. 이것을 시험하는 가장 쉬운 방법은 연결을 설정하고 서버를 네트워크로부터 분리시킨 후에 서버 호스트를 정지시킨다. 서버를 다시 재시동하여 서버 호스트를 네트워크에 연결시킨다. 이때 클라이언트는 서버가 정지되었음을 알지 못한다.

앞 절에서 이야기한 대로 실제로 서버 호스트가 정지되었을 때 클라이언트가 서버에게 데이터를 보내지 않으면 클라이언트는 서버 호스트가 정지되었음을 알지 못한다. 다음 같은 단계를 따른다.

1. 서버를 시작한 후 클라이언트를 시작한다. 연결이 되었는지를 확인하기 위해 한 줄을 입력한다.
2. 서버 호스트를 정지하고 재시동한다.
3. 서버 호스트에 TCP 데이터 세그먼트로 보낼 한 줄을 클라이언트에 입력한다.
4. 서버 호스트가 정지한 후 재시동하면 정지 전에 있었던 연결에 관한 모든 정보를 TCP는 잃게 된다. 따라서 서버 TCP는 클라이언트로부터 받은 데이터 세그먼트에 대한 답신으로 RST를 보낸다.
5. RST를 받을 때 클라이언트는 readline 호출에서 블록되어 있으므로, readline은 ECONNRESET 오류를 돌려준다.

만약 클라이언트가 능동적으로 데이터를 보내지 않고서도 서버 호스트의 장애를 알아내는 것이 중요하다면 다른 기법이 필요하다.

C. 서버 Host의 Shutdown

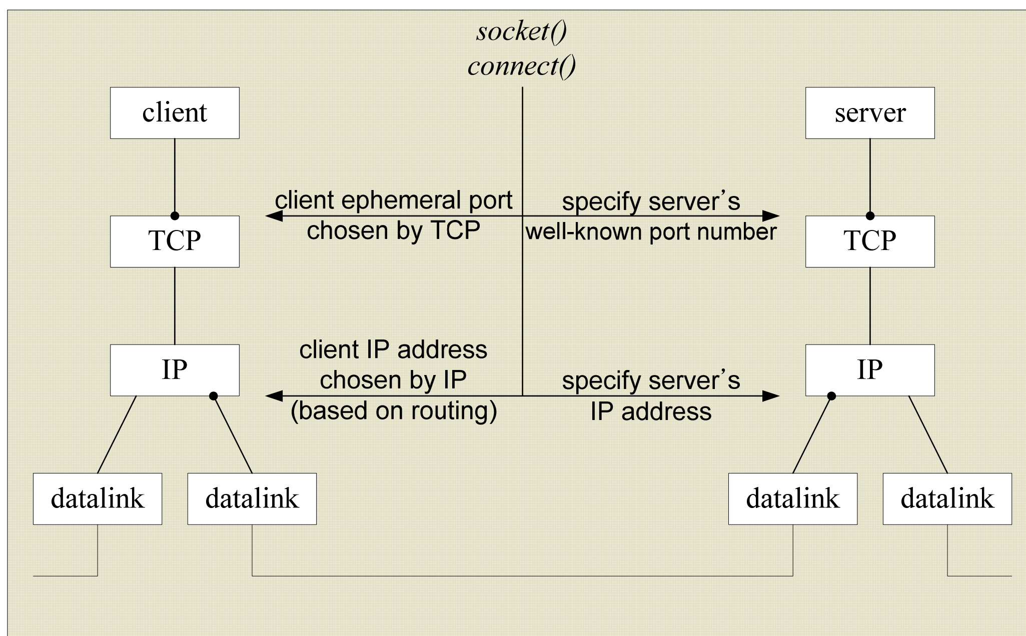
앞의 두 절에서 서버 호스트의 장애 발생과 네트워크를 통해 도달할 수 없는 서버 호스트를 다루었다. 이 절에서는 호스트에서 서버 프로세스가 동작 중일 때 서버 호스트를 운영자가 끈다면 어떤 일이 일어나는지를 생각한다.

유닉스 시스템이 종료할 때, init프로세스는 보통 SIGTERM 신호를 모든 프로세스에게 보내고 (이

신호를 포착할 수 있다) 일정 시간 동안 기다린 후(보통 5~20초) 아직 실행 중인 모든 프로세스에게 SIGKILL 신호 (이 신호는 포착할 수 없다)를 보낸다. 이것은 모든 실행 중인 프로세스에게 정리하고 종료하도록 짧은 시간을 준다. 만약 서버가 SIGTERM을 포착하여 종료하지 못하면 SIGKILL 신호에 의해 종료된다. 프로세스가 종료할 때 모든 open descriptor는 닫힌다. 앞서 언급한 대로, 클라이언트가 서버 프로세스의 종료를 곧바로 알기 위해서는 클라이언트에서 select 또는 poll 함수를 사용해야만 한다.

5.8 TCP 예제 요약

모든 TCP 클라이언트와 서버는 서로 통신하기 이전에 연결에 대한 socket pair를 규정해야 한다: 이는 local IP 주소와 local 포트, 원격지 IP 주소와 원격지 포트로 그림 5.5에서 굵은 점으로 표시한다. 이 그림은 클라이언트 관점에서 그렸다. 원격지 IP주소와 포트는 클라이언트가 호출하는 connect에서 규정해야 한다. 두 개의 local 값은 보통 connect 함수의 일부로 커널이 선택한다. 클라이언트가 connect 이전의 bind 호출에서 이들 local 값을 규정하는 옵션이 있으나 특별한 경우이다.

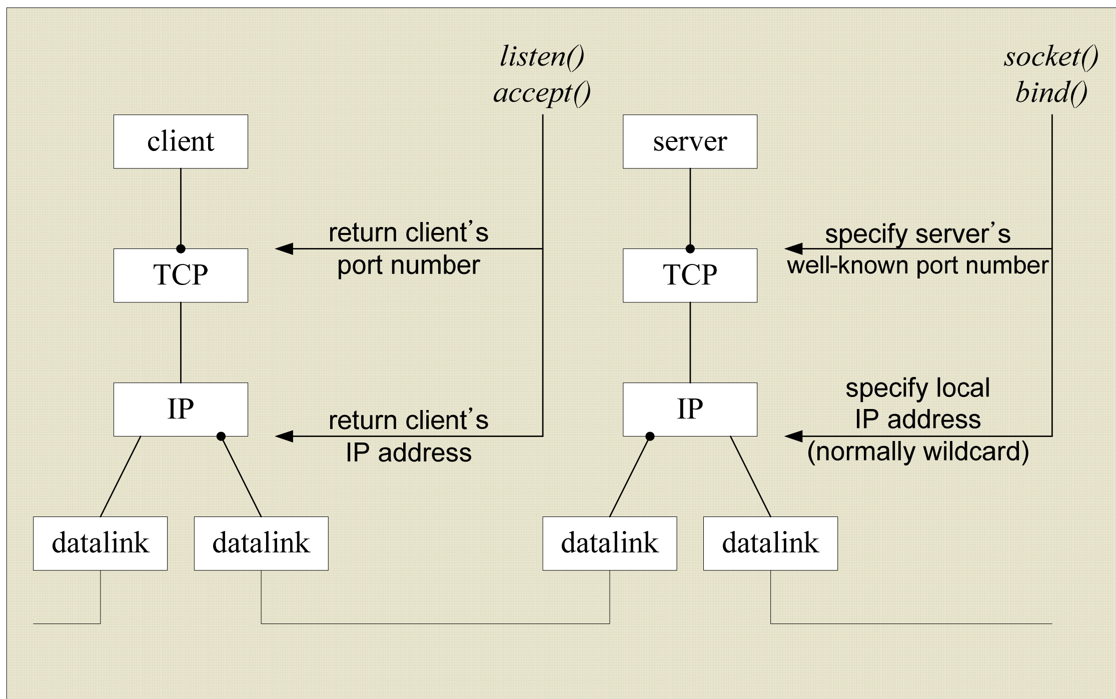


<그림 5.5> TCP Client/Server 예제: Client 관점

4절에서 언급했던 것처럼, 클라이언트는 연결이 이루어진 후에 getsocketname을 호출하여 커널이 선택한 두 local 값을 얻을 수 있다.

그림 5.6에서는 서버측에서 본 동일한 네 개의 값을 보여준다.

local 포트(서버의 well-known port)는 bind에서 규정한다. 서버는 임의 IP주소가 아닌 것을 묶어 특정 local 접속에 대한 연결만을 받도록 제한할 수 있지만, 보통은 이 호출에서 임의 IP 주소를 규정한다. 만약 서버가 여러 네트워크에 연결된 호스트에서 임의 IP 주소를 묶으면 연결이 이루어진 후에 getsockname을 호출하여 local IP 주소를 알 수 있다. 두 원격지 값은 accept가 서버에게 돌려준다. accept를 호출하는 서버가 exec한 다른 프로그램은 클라이언트의 IP 주소와 포트를 알려면 getpeername을 호출할 수 있다.



<그림 5.6> TCP Client/Server 예제: 서버 관점

5.9 USER DATA FORMAT

예제에서 서버는 클라이언트로부터 받은 요청을 시험하지 않는다. 서버는 그저 newline까지의 데이터를 읽어서 그대로 클라이언트에게 돌려보내고 다시 다음 newline 문자를 찾는다. 하지만 대개는 클라이언트와 서버 사이에서 오가는 데이터의 형식에 관심을 두어야 한다.

Example: Passing Text Strings between Client and Server

이전처럼 클라이언트로부터 문자열을 읽지만 여백 문자로 구분되는 두 정수를 가진 줄을 읽어 이들의 합을 돌려주도록 서버를 수정한다. 클라이언트와 서버의 main 함수와 str_cli 함수는 그대로 두고 바뀌는 것은 str_echo 함수로 다음 예제에서 볼 수 있다.

<str_ech08.c>

```
----------------------------------------------------------------------------
 1 #include    "unp.h"

 2 void
 3 str_echo(int sockfd)
 4 {
 5    long    arg1,    arg2;
 6    ssize_t n;
 7    char    line[MAXLINE];

 8    for ( ; ; ) {
 9       if ( (n = Readline(sockfd, line, MAXLINE)) == 0)
10          return;           /* connection closed by other end */

11       if (sscanf(line, "%ld%ld", &arg1, &arg2) == 2)
12          snprintf(line, sizeof(line), "%ld\n", arg1 + arg2);
13       else
14          snprintf(line, sizeof(line), "input error\n");

15       n = strlen(line);
16       Writen(sockfd, line, n);
17    }
18 }
----------------------------------------------------------------------------
```

11-14

문자열의 두 인수를 긴 정수로 변환하려고 sscanf를 호출하며, 결과를 문자열로 다시 바꾸려고 snprintf를 호출한다. 이 새로운 클라이언트와 서버는 클라이언트와 서버 호스트의 바이트 순서에 관계없이 잘 동작한다.

Example: Passing Binary Structures between Client and Server

여기서는 소켓을 통하여 문자열 대신에 이진 값을 교환하도록 클라이언트와 서버를 수정한다. 다른 바이트 순서를 가지는 호스트에서 또는 긴 정수의 크기가 다른 호스트에서 클라이언트와 서버를 실행할 때는 제대로 작동하지 않는다는 것을 알 수 있다.

클라이언트와 서버의 main 함수는 바뀌지 않는다. 두 인수를 갖는 구조와 결과를 갖는 다른 구조를 다음 예제같이 정의하여 sum.h에 넣는다. 추가 예제에서는 str_cli 함수를 보여준다.

```
<sum.h>
----------------------------------------------------------------------------
1 struct args {
2    long   arg1;
3    long   arg2;
4 };

5 struct result {
6    long   sum;
7 };
----------------------------------------------------------------------------
```

```
<str_cli09.c>
----------------------------------------------------------------------------
1 #include     "unp.h"
2 #include     "sum.h"

3 void
4 str_cli(FILE *fp, int sockfd)
5 {
6    char    sendline[MAXLINE];
7    struct args args;
8    struct result result;

9    while  (Fgets(sendline, MAXLINE, fp) != NULL) {

10        if  (sscanf(sendline, "%ld%ld", &args.arg1, &args.arg2) != 2) {
11            printf("invalid input: %s", sendline);
12            continue;
13       }
14       Writen(sockfd, &args, sizeof(args));

15       if (Readn(sockfd, &result, sizeof(result)) == 0)
16           err_quit("str_cli: server terminated prematurely");

17       printf("%ld\n", result.sum);
18    }
19 }
----------------------------------------------------------------------------
```

다음 예제에서는 str_echo 함수를 보여준다.

```
<str_ech09.c>
--------------------------------------------------------------------------
 1 #include     "unp.h"
 2 #include     "sum.h"

 3 void
 4 str_echo(int sockfd)
 5 {
 6    ssize_t n;
 7    struct args args;
 8    struct result result;

 9    for ( ; ; ) {
10       if ( (n = Readn(sockfd, &args, sizeof(args))) == 0)
11          return;           /* connection closed by other end */

12       result.sum = args.arg1 + args.arg2;
13       Writen(sockfd, &result, sizeof (result));
14    }
15 }
--------------------------------------------------------------------------
```

같은 구조의 두 컴퓨터에서 서버를 실행하면 모든 동작은 순조롭다. 다음은 클라이언트 측에서의 상호작용이다. 다음은 상기 <tcpcli09.c> 예제의 실행화면이다.

```
[LeeDonghwa@localhost tcpcliserv]$ ./tcpcli09 155.230.105.166
11 22
33
-11 -44
-55
```

그러나 다른 구조의 두 컴퓨터에서 클라이언트와 서버를 실행하면, 제대로 작동하지 않는다.

```
linux % tcpcli09 206.168.112.96
1 2                        we type this
3                          and it works
-22 -77                    then we type this
-16777314                  and it does not work
```

문제는 클라이언트가 두 정수를 소켓을 통하여 little-endian format으로 보내고, 서버는 이를 big-endian integers로 해석하는 데 있다. 위에서 양의 정수는 작동하지만 음의 정수는 실패하는 것을 알 수 있다. 이 예제에서 잠재적인 문제가 셋 있다.

1. 구현에 따라 이진수를 다른 형식으로 저장한다. Section3.4에서 설명한 대로 가장 일반적인 형식은 big-endian과 little-endian 형식이다.

2. 구현에 따라 동일한 C 데이터형을 다르게 저장할 수 있다. 예를 들면 대부분의 32비트 유닉스 시스템은 long에 대해 32비트를 사용하지만 64비트 시스템은 64비트를 사용한다. short, int, long에 대해서 정해진 크기를 가진다고 보장할 수 없다.

3. 구현에 따라 그 컴퓨터의 배열 제한과 여러 가지 데이터형에 대해 사용하는 비트수에 따라 구조를 다르게 취급한다. 소켓을 통하여 이진 구조를 보내는 것은 결코 현명한 방법이 아니다.

이 데이터 형식 문제에 대한 두 가지의 해결 방안이 있다.

1. 모든 숫자 데이터를 문자열로 보낸다. 두 호스트가 동일 문자 집합을 가진다고 가정한다.

2. 지원되는 데이터형(비트 개수, big-endian or little-endian)의 이진 형식을 명시적으로 정의하고, 클라이언트와 서버 사이에 모든 데이터를 이 형식으로 주고 받는다. RPC (Remote Procedure Call) 패키지는 통상 이 기법을 사용한다.

6. I/O MULTIPLEXING: SELECT()

이전 장에서, 우리는 동시에 두 가지 입력을 다루는 TCP 클라이언트를 보았다: 표준 입력과 TCP 소켓. 우리는 클라이언트가 서버 프로세스가 죽고 fgets(표준 입력)을 호출하는 것을 막았을 때, 문제를 언급했다. 서버 TCP는 정확히 FIN을 클라이언트 TCP에 보냈다, 그러나 클라이언트 프로세스가 표준 입력을 읽는 것이 막혀서 소켓으로부터 그것을 읽을 때까지 EOF를 절대 볼 수 없었다. 우리가 필요한 것은 우리가 하나 혹은 더 많은 입/출력 조건이 준비되었는지 아닌지를 알려주기를 원하는 kernel에게 특성을 말해주는 것이다. (예를 들면, 입력이 읽혀질 준비가 되거나, 구분자가 더 많은 출력을 가지도록 하는 것). 이 특성을 입/출력 다중 송신이라 불리며 select 함수에 의해서 제공된다.

복합 입/출력은 전형적으로 다음 시나리오에서 네트워크 어플리케이션으로 사용되었다:

- 클라이언트가 복합적인 descriptor를 다룰 때(일반적으로 양방향 입력과 네트워크 소켓), 복합적인 입/출력은 사용되어야 한다. 이것은 우리가 이전에 언급했던 시나리오이다.
- 희박하지만, 클라이언트가 동시에 복합적인 소켓을 다룰 때 가능하다.
- 만약 TCP 서버가 listening 소켓과 그것과 연결된 소켓 모두 다룬다면, 복합적인 입/출력은 일반적으로 사용된다.
- 만약 서버가 TCP와 UDP 모두 다룬다면, 복합적인 입/출력은 일반적으로 사용된다.
- 만약 서버가 복합적인 서비스와 복합적인 프로토콜을 다룬다면, 복합적인 입/출력이 일반적으로 사용된다.

복합적인 입/출력은 네트워크 프로그래밍에서 제한이 없다. 많은 중요한 어플리케이션이 이러한 기술을 필요로 한다.

6.1 I/O 모델

Select를 설명하기 전에, 우리는 되돌아가서 Unix에서 우리가 사용할 수 있는 다섯 개의 다른 입/출력 모델의 기본적인 차이점을 아는, 더 큰 그림을 볼 필요가 있다.

- blocking 입/출력
- nonblocking 입/출력
- 복합적인 입/출력(select와 poll)
- 신호가 전달되는 입/출력 (SIGIO)
- 비동기식 입/출력 (POSIX aio_함수)

우리는 이 섹션에서 모든 예제에서 일반적으로 입력 동작을 위한 두 가지 전혀 다른 양상이 있는 것을 보여준다:

1. 준비가 되기까지 데이터를 기다리는 것
2. kernel에서 process로 데이터 복사

소켓에서 입력 동작을 위해, 첫 번째 단계는 일반적으로 네트워크 상에서 데이터가 도착하기를 기다리는 것을 포함하고 있다. packet이 도착하면, kernel안에서 버퍼로 복사된다. 두 번째 단계는 이 데이터를 kernel의 버버퍼로부터 우리의 어플리케이션 버퍼로 복사하는 것이다.

Blocking 입/출력 모델

입/출력에 대한 가장 일반적인 모델은 blocking 입/출력 모델로, 책에서 이 때까지 사용된 모든 예제가 해당한다. 디폴트로, 모든 소켓은 blocking이다. 우리의 예제에 대한 datagram 소켓이 사용하는 것으로, 우리는 Figure 6.1에서 보여주고 있는 시나리오이다.

우리는 이 예제를 위해 TCP 대신에 UDP를 사용한다. 왜냐하면 "준비상태"로 읽혀질 데이터에 대한 개념이 간단하다: 전체적인 datagram이 받아지거나 안 받아진다. TCP는 좀 더 복잡하다. 추가적인 변수(이를테면 소켓의 최저상태)가 활동하기 시작한다.

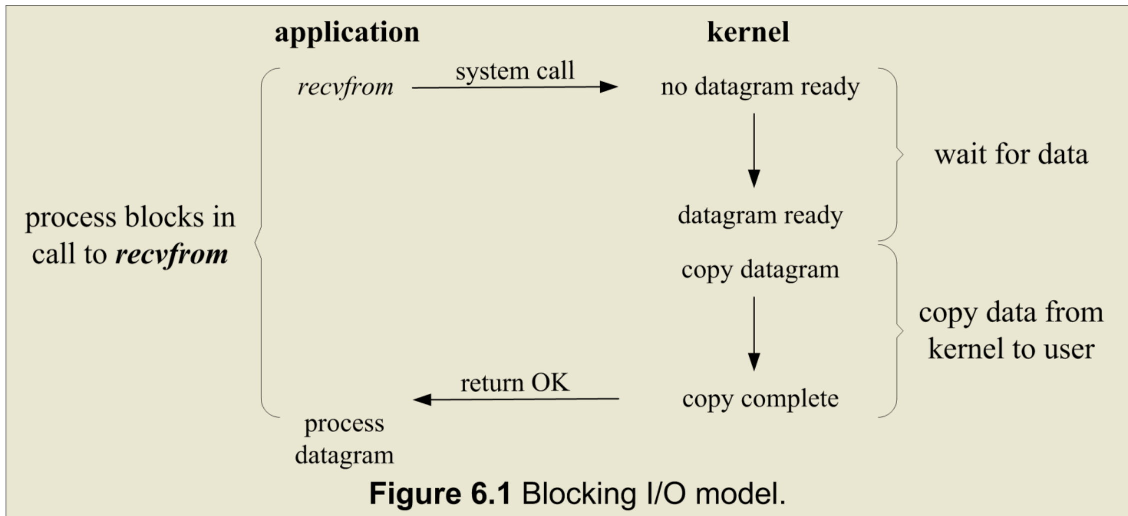


Figure 6.1 Blocking I/O model.

이 절의 예제에서, 우리는 역시 recvfrom을 시스템 콜처럼 부른다. 왜냐하면 우리는 우리의 어플리케이션과 kernel사이를 구별하기 때문이다. recvfrom이 구현된 것에 관계없이, 일반적으로 어플리케이션에서 동작하는 것에서 kernel에서 동작하는 것으로, 때에 따라서는 늦을 경우 어플리케이션에서 리턴할 때까지, 바뀐다.

그림 6.1에서, process는 recvfrom을 호출하고 system call은 datagram이 도착할 때 까지 반환되지 않고 우리의 어플리케이션 버퍼로 복사되거나 에러가 발생한다. 대부분의 일반적인 에러는, 섹션 5절에서 이야기한 신호에 의해 interrupt되는 시스템 콜이다. 우리는 우리의 process가 recvfrom을 호출하고 반환할 때까지 계속해서 block되어있다고 말한다. recvfrom이 성공적으로 리턴할 때 우리의 어플리케이션은 datagram을 처리한다.

Nonblocking 입/출력 모델

우리는 소켓을 막히지 않도록 설정할 때, 우리는 kernel에 말하고 있다 "내가 요구하는 입/출력 동작은 process를 sleep하지 않고 완료될 수 없고, process를 sleep하지 않는다, 그러나 대신에 에러를 반환한다. 우리는 nonblocking 입/출력을 챕터 16에서 언급할 것이다. 그러나 Figure 6.2는 우리가 고려하고 있는 예제들을 요약해서 보여준다.

우선 반환하는 데이터가 없는 recvfrom함수를 세 번째로 호출한다. 그러면 kernel은 대신에 EWOULDBLOCK에 대한 오류를 반환한다. datagram이 준비된 recvfrom함수를 네 번째로 호출한다. 그리고 우리의 어플리케이션 버퍼가 복사되고, recvfrom은 성공적으로 반환한다. 그러면 우리는 데이터를 처리한다.

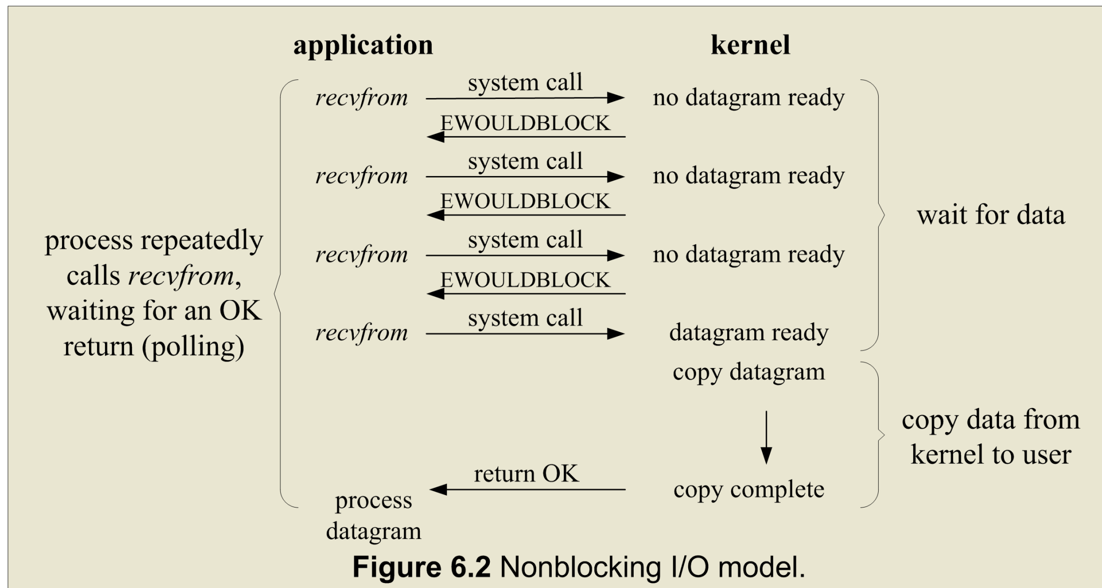


Figure 6.2 Nonblocking I/O model.

어플리케이션이 이처럼 nonblocking descriptor에서 recvfrom을 호출하는 루프를 대신할 때 polling이라 부른다. 어플리케이션은 만약 몇몇 동작이 준비된다면 보이는 것처럼 kernel을 계속해서 pollng한다. 이것은 종종 CPU 시간을 낭비하거나, 이 모델은 가끔 충돌이 일어난다, 일반적으로 시스템 상에서 하나의 함수로 작동한다.

다중 입/출력 모델

다중 입/출력으로, 우리는 select 또는 poll을 호출하고 이 두 가지 시스템 콜 중의 하나를 막는다. 대신에 실제 입/출력 시스템 콜을 막는다. 그림 6.3은 다중 입/출력 모델을 요약한다.

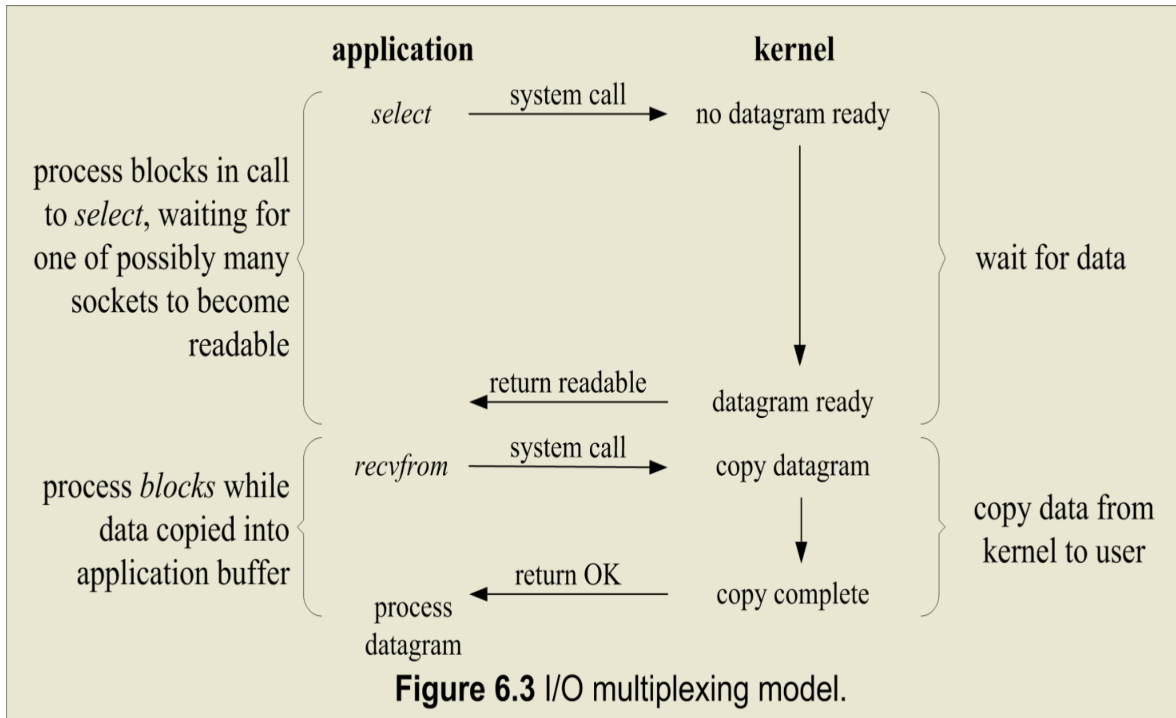


Figure 6.3 I/O multiplexing model.

우리는 select함수의 호출을 막고, datagram socket이 읽을 수 있게 기다린다. select함수가 읽을 수 있는 socket을 반환할 때, 그러면 우리는 어플리케이션 버퍼에서 datagram을 복사하는 recvfrom을 호출한다.

그림 6.1과 6.3을 비교하면, 별 다른 이점이 나타나지 않는다. 그리고 사실 약간의 손해가 존재한다. 왜냐하면 사용하고 있는 select함수는 하나가 아닌 두 개의 시스템 콜을 요구한다. 그리고 우리는 이 장의 뒷부분에서 볼 것이다. 우리는 하나 이상의 descriptor가 준비되기를 기다릴 수 있다.

이와 달리 접근해 보면 입/출력 모델은 blocking 입/출력으로 멀티쓰레딩을 사용한다. 그 모델은 select함수를 사용하는 대신에 다중 파일 descriptor를 막는 것 그리고 프로그램은 다중 쓰레드를 사용하는 것과 각각의 쓰레드는 recvfrom같이 시스템 콜들을 막기 위해 호출하는 것을 제외하고, 위에서 언급한 모델과 매우 유사하다.

신호-처리 입/출력 모델

우리는 descriptor가 준비가 됐을 때 SIGIO 신호로 kernel에 알리기 위해 신호를 사용할 수도 있다. 우리는 이를 신호-처리 입/출력이라 부르고 그림 6.4에서 이를 요약해서 보여준다.

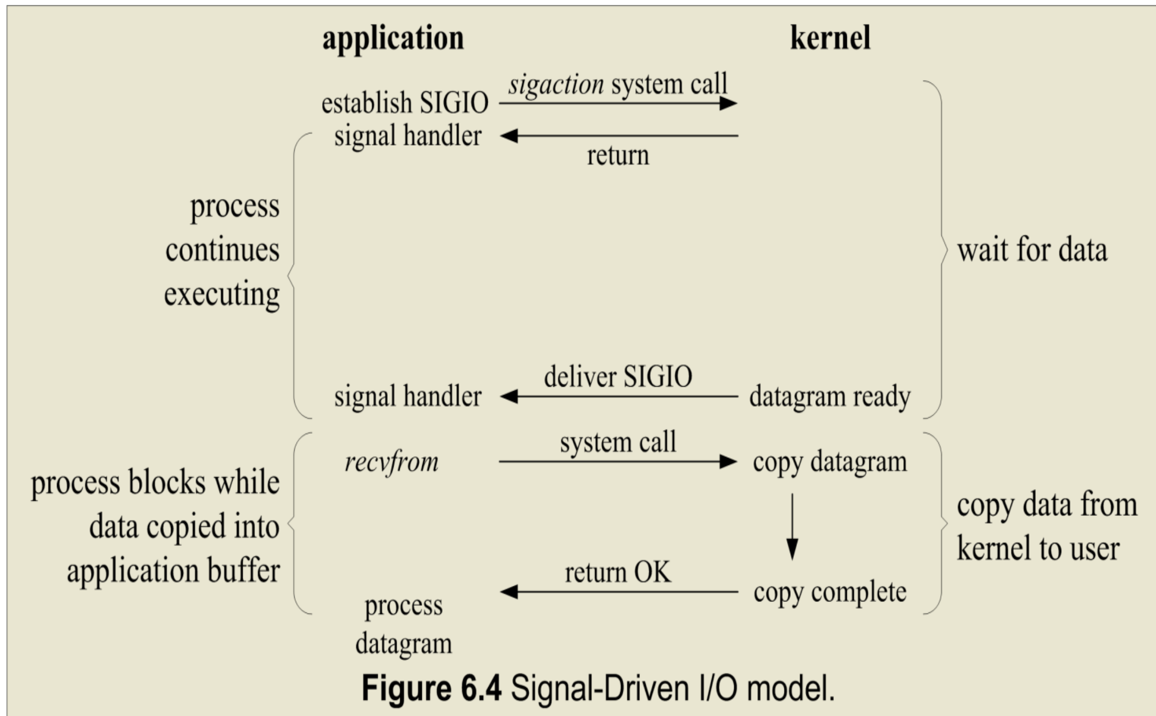


Figure 6.4 Signal-Driven I/O model.

우리는 우선 신호-처리 입/출력을 위한 소켓을 사용할 수 있다. 그리고 sigaction 시스템 콜을 사용하는 신호 처리기를 설치한다. 이 시스템 콜로부터의 반환 값은 즉시 발생하고 우리의 프로세스가 계속된다. 다시 말해 중단되지 않는다. datagram이 읽을 준비가 되면, SIGIO가 우리의 프로세스를 위해 생성된다. 이와는 달리 우리는 recvfrom을 호출함으로써 시그널 핸들러를 통해 datagram을 읽을 수 있고, main 루프는 데이터가 처리될 준비가 됐다는 것을 알게 되거나, 우리는 main 루프가 알 수 있고, datagram을 읽도록 할 것이다.

우리가 시그널을 어떻게 처리하던지, 이 모델의 장점은 datagram이 도착하기를 기다리는 동안 블록이 없다는 것이다. main 루프는 계속 실행 중이고, 단지 데이터가 처리될 준비가 되거나, 읽을 준비가 되는지를 기다리는 시그널 핸들러에 의해서 알려지기를 기다린다.

비동기 입/출력 모델

비동기 입/출력 모델은 POSIX 표준에 의해 정의되며, 현재의 POSIX 표준과 융화된 다양한 표준에서 나타난 실시간 함수에서 다양한 차이가 있다. 일반적으로, 이 함수들은 kernel이 작동 시작을 말함으로써 작동하고, 우리는 모든 동작(kernel에서 우리의 버퍼로 데이터가 복사하는 것을 포함)이 끝날 때 알 수 있다. 이 모델과 이 전 섹션에서 이야기한 신호-처리 입/출력 모델의 주된 차이는 신호-처리 입/출력 모델은 kernel이 입/출력 작동이 초기화 될 경우 우리에게 이야기를 해주는 것이나, 비동기 입/출력 모델은 kernel이 입/출력 작동이 완료된 경우 우리에게 이야기를 해준다. 그림 6.5에서 예를 볼 수 있다.

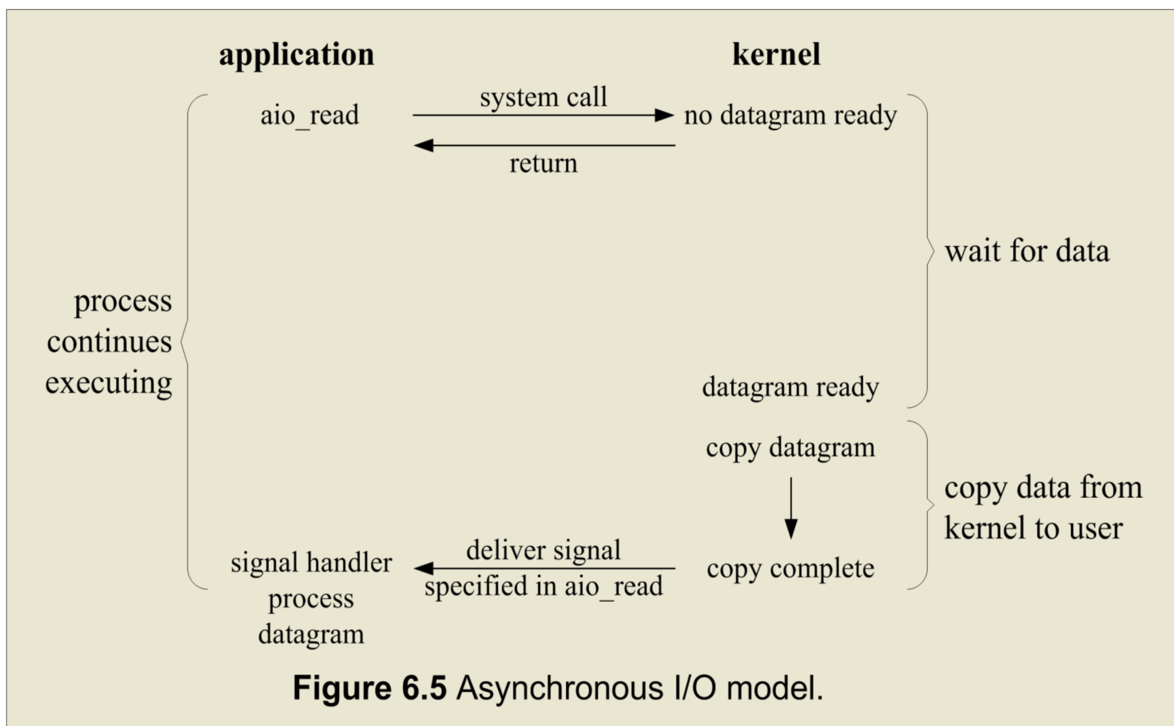


Figure 6.5 Asynchronous I/O model.

우리는 aio_read(POSIX 비동기 입/출력 함수들은 aio_ 또는 lio_로 시작한다)를 호출하고 descriptor, 버퍼 포인터, 버퍼 크기(read를 위한 같은 세가지 인자), 파일 offset(lseek와 유사)이 kernel을 통과하고, 그리고 우리는 모든 작동이 끝난 후 어떻게 알아 차리는가? 이 시스템 콜은 즉시 리턴을 하고, 우리의 프로세스는 입/출력이 끝나기를 기다리는 동안 블록되지 않는다. 우리는 이 예제에서 우리가 kernel이 동작이 끝났을 때, 몇몇 시그널을 생성하기를 요청한다고 가정한다. 이 시그널은 데이터가 우리의 어플리케이션 버퍼로 복사될 때까지 생성되지 않는다. 이것이 신호-처리 입/출력 모델과의 차이점이다.

비동기 입/출력 모델

그림 6.6은 다섯 개의 다른 입/출력 모델에 대한 비교이다. 첫 번째와 네 번째 모델 사이의 주된 차이점은 우선 네 모델은 네 모델이 같은 두 번째 단계: 프로세스는 데이터가 kernel에서 호출자의 버퍼로 복사되는 동안 recvfrom이 호출될 때 블록된다.

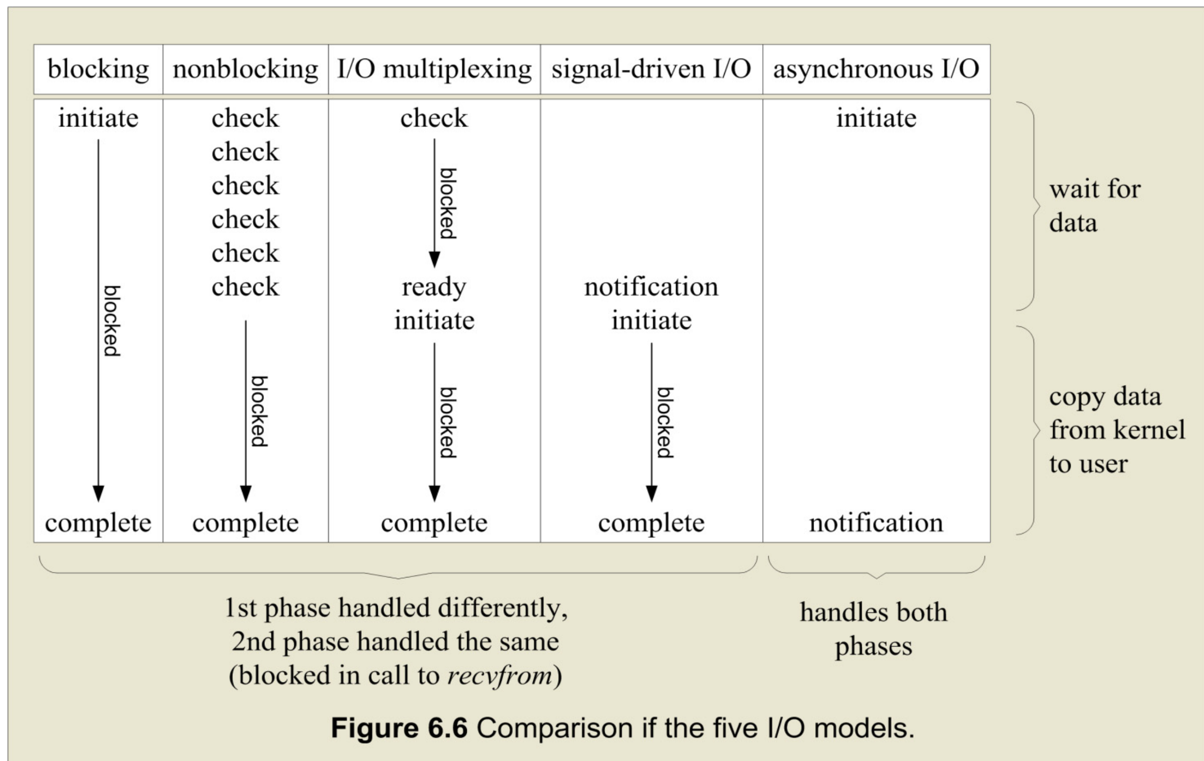


Figure 6.6 Comparison if the five I/O models.

동기 입/출력 모델 대 비동기 입/출력 모델

POSIX는 이 두 가지 용어를 다음과 같이 정의한다.

- 동기 입/출력 동작은 입/출력 동작이 끝날 때까지 프로세스 요청을 막는 것을 말한다.
- 비동기 입/출력 동작은 프로세스 요청을 막지 않는 것을 말한다.

이 정의에서 보면, 처음 네 가지 모델-blocking, nonblocking, 다중 입/출력, 신호-처리 입/출력-은 모두 동기 입/출력 모델이다. 왜냐하면 실제 입/출력 동작(recvfrom)은 프로세스를 막는다. 단지 비동기 입/출력 모델만이 비동기 입/출력 정의와 부합한다.

6.2 SELECT 함수

이 함수는 kernel이 발생하는 다양한 이벤트 중에 어떤 하나를 기다리는 것을 알리고 이 이벤트들의 하나 또는 그 이상 발생했을 경우, 또는 정해진 시간이 지난 뒤 반환된다. 예를 들면, select를 호출하는 경우, 다음의 조건을 만족하면 반환된다:

- 집합{1, 4, 5}에서 descriptor 중에 몇몇이 읽을 준비가 됐을 때 (readable)
- 집합{1, 4, 5}에서 descriptor 중에 몇몇이 쓸 준비가 됐을 때 (writable)
- 집합{1, 4, 5}에서 descriptor 중에 몇몇이 미결정 예외 조건을 가질 때 (exceptable)
- 정해진 시간이 지났을 때

즉, 우리는 우리가 관심있는 descriptor(읽기, 쓰기, 또는 예외 조건에 대한 것)가 무엇인지, 얼마나 시간이 경과했는지를 kernel에 이야기 할 수 있다. 우리가 관심이 있는 descriptor는 소켓에 제한 받지 않는다; 어떤 descriptor의 경우 select를 사용하는 테스트가 될 수 있다.

```
#include <sys/select.h>

#include <sys/time.h>

int select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptest, const struct timeval *timeout);

    Returns:positive count of ready descriptors, 0 on timeout, -1 on error
```

우리는 이 함수에서의 우리의 descriptor가 함수의 마지막 인자로 시작한다, 그리고 그 descriptor는 준비된 특정 descriptor중의 하나를 얼마나 기다려야 하는지 말해준다. timeval 구조체는 세컨드와 마이크로 세컨드의 크기를 지정한다.

```
struct timeval {
        long   tv_sec;       /* seconds */
        long   tv_usec;            /* microseconds */
};
```

세 가지 가능성이 있다.

1. 영원히 기다린다: 단지 특정한 descriptor중의 하나가 입/출력이 준비됐을 때 리턴한다. 이를 위해, 우리는 null 포인터로써 timeout인자를 열거한다.

2. 정해진 시간만큼 기다린다: 단지 특정한 descriptor중의 하나가 입/출력이 준비됐을 때 리턴한다. 그러나 timeout인자로써 지정된 구조체 timeval에서 특정한 마이크로 세컨드와 세컨드를 지나서 기다리지는 않는다.

3. 전혀 기다리지 않는다: descriptor를 체크한 후 바로 리턴한다. 이것은 polling이라 불린다. 이것은 특별하다, timeout인자는 0임에 틀림 없는 타이머 값(구조체에 의해 특정한 마이크로 세컨드와 세컨드)과 timeval 구조체를 정해야 한다.

첫 번째 두 시나리오에서 기다리는 것은 만약 프로세스가 signal을 캐치하고 signal handler로부터 반환한다면 일반적으로 인터럽트된다.

비록 timeval 구조체가 우리에게 수 마이크로 세컨드 내로 결정을 조건으로 지정시키지만, 실제로 kernel에서 지원하는 결정은 가끔 더 정확하지 않을 경우도 있다. 예를 들면, 많은 Unix kernel들은 수 십 초 이상 시간 초과를 하기도 한다. 역시 향상된 latency 스케줄링이 필요하다, 이는 kernel이 이 process가 동작하는 것을 스케줄링 하기 전에 timer가 시간 초과된 후 약간의 시간이 지나는 것을 의미한다.

몇몇 system상에서, select는 만약 시간초과시 tv_sec 필드가 1억 초 이상 지난다면 EINVAL 실패가 될 것이다. 물론, 저것은 매우 큰 시간초과이고(3년 이상) 매우 유용하지 않다. 그러나 요점은 timeval 구조체는 select에 의해 지원되지 않는 값을 표현할 수 있다.

Timeout 인자에서 const qualifier는 return시 select에 의해 수정되지 않는 것을 의미한다. 예를 들면, 만약 우리가 10초의 시간제한을 두고, select는 시간초과 전에 준비된 descriptor를 하나 혹은 그 이상을 return하거나, EINTR에 대한 error를 return한다면, timeval 구조체는 함수가 return할 때 남는 수 많은 시간이 갱신되지 않는다. 만약 우리가 이 값을 알기를 원한다면, 우리는 select를 호출하기 전 시간과 그것이 다시 return하는 시간을 알아야 한다.

몇몇 Linux 버전은 timeval구조체를 수정한다. 그러므로, 예를 들면, timeval구조체가 return시에 정듸되지 않았다고 가정을 하고, select를 호출하기 전에 그것을 초기화 한다. POSIX는 const qualifier를 구체화한다.

세 개의 중간 인자(readset, writeset, exceptest)는 우리가 kernel이 읽기, 쓰기, 예외 조건을 테스트 하기 위해 원하는 descriptor를 구체화한다. 현재 지원하는 단 두 가지 예외 조건이 있다.

1. Socket을 위한 out-of-band의 도착.
2. Pseudo-terminal의 master측면에서 읽혀진 Control 상태 정보에 대한 존재는 packet 모드에서 개정된다. 이 책에서는 pseudo-terminal에 대해서 언급하지 않는다.

문제는 이 세 가지 인자에 대한 하나 혹은 그 이상의 descriptor 값을 어떻게 구체화 할 것 인지 이다. Select는 descriptor에 따라서 각각의 정수형에 각각의 bit로, 전형적으로 정수형 배열, descriptor set을 사용한다. 예를 들면, 32-bit 정수형을 사용하는, 0에서 31까지 descriptor에 따른 배열의 첫 번째 구성 요소이고, 32부터 63까지는 descriptor에 따른 배열의 두 번째 구성 요소이고, 그런 식으로 되어있다. 모든 상세 구현은 application에 무관하고, 다음 네 가지 macro와 fd_set datatype을 숨기고 있다.

```
void FD_ZERO(fd_set *fdset);            /* clear all bits in fdset */
void FD_SET(int fd, fd_set *fdset); /* turn on the bit for fd in fdset */
void FD_CLR(int fd, fd_set *fdset); /* turn off the bit for fd in fdset */
void FD_ISSET(int fd, fd_set *fdset);/* is the bit for fd on in fdset ? */
```

fd_set type에 대한 가변적인 정의와 descriptor 1, 4, 5를 위해 bit를 바꾸어보자.

```
fd_set rset;
FD_ZERO(&rset);     /* initialize the set: all bits off */
FD_SET(1, &rset);   /* turn on bit for fd 1 */
FD_SET(4, &rset);   /* turn on bit for fd 4 */
FD_SET(5, &rset);   /* turn on bit for fd 5 */
```

중요한 것은 set을 초기화 하는 것이다. 왜냐하면 만약 set이 자동적으로 변수에 할당되지 않거나, 초기화 되지 않으면, 예측할 수 없는 결과가 발생할 수 있기 때문이다.

select는 readset, writeset, exceptset 포인터에 의해 지적되는 descriptor set을 수정한다. 이 세가지 인자는 value-result 인자이다. 우리가 함수를 호출할 때, 우리는 우리가 관심 있는 descriptor에 대한 값을 구체화하고, return시에는 descriptor가 준비됐다는 것을 가리키는 결과를 구체화한다. 우리는 return시 fd_set 구조체에서 특정한 descriptor를 테스트 하기 위해 FD_ISSET macro를 사용한다. 어떤 return시 준비가 안된 descriptor는 descriptor set에서 그것과 연관된 bit를 제거할 것이다. 이를 처리하기 위해서, 우리는 우리가 select를 호출하는 매 시간마다 관심 있는 모든 descriptor set에 대한 모든 bit를 바꾼다.

select를 사용할 때 두 가지 가장 흔한 프로그래밍 error는 가장 큰 descriptor 수에 1을 더하는 것을 잊어버리는 것과 descriptor set이 value-result 인자라는 것을 잊어버리는 것이다. 두 번째 error는 우리가 1로 생각하고 있을 때, descriptor set에서 0으로 정해진 bit로 호출된 select 결과이다.

이 함수로부터 return값은 모든 descriptor set에 걸쳐서 준비된 모든 bit의 수를 가리킨다. 만약 timer값이 어떤 descriptor가 준비되기 전에 만기된다면, 0이 return될 것이다. return 값이 -1이라면 error를 가리킨다.

Descriptor의 준비 조건은 무엇인가?

select가 socket을 위해 "ready"를 return하는 조건에 대해서 알아보자.

1. 만약 어떤 다음 네 가지 조건이 사실이라면 socket은 읽기 위한 준비 상태이다.

   a. socket receive buffer에서 data에 대한 byte의 수는 socket receive buffer를 위한 현재 크기의 가장 작은 크기보다 크거나 같아야 한다. socket에서 읽기 동작은 block되지 않을 것이며, 0보다 큰 값(예를 들면, 준비된 data)으로 return될 것이다. 우리는 SO_RCVLOWAT socket opton을 사용함으로써 이 가장 작은 값을 정할 수 있다. 그것은 TCP와 UDP socket을 위해 1이 기본값으로 설정된다.

   b. read half에 대한 연결이 닫혔을 때(예를 들면, FIN을 받고 있는 TCP연결). socket에서 읽기 동작은 block되지 않을 것이고, 0이 return될 것이다(예를 들면, EOF).

   c. socket은 listening socket이고, 종료된 연결의 수는 0이 아니다. listening socket에서 accept는 일반적으로 block되지 않을 것이다.

d. socket error는 해결하지 못하고 있다. socket에서 읽기 동작은 block되지 않을 것이고, 특정한 error 조건에서 설정된 errno으로 error (-1)을 리턴할 것이다. 이 pending error들은 getsockopt를 호출하고 SO_ERROR socket option을 제시함으로써 제거되거나 불려질 수 있다.

2. socket은 만약 다음 조건 중 어떤 것이 사실이라면 writing을 준비 상태이다.

   a. socket send buffer에서 이용 가능한 공간에 대한 byte 수는 socket send buffer와 다른 것: (i) socket이 연결되거나, (ii)연결 요청이 없는 socket(예를 들면 UDP), 을 위한 가장 낮은 부분의 현재 크기와 같거나 크다. 이것은 만약 우리가 socket을 block하지 않도록 설정한다면, 쓰기 동작이 block되지 않을 것이고, 양수 값(예를 들면, 전송 계층에 의해 연결된 byte의 수)을 return할 것이다. 우리는 SO_SNDLOWAT socket option을 사용하는 최저점을 설정한다. 이 최저점은 일반적으로 TCP와 UDP socket을 위해 2048을 기본값으로 한다.

   b. write half에 대한 연결이 닫혀 있을 때. socket에서 쓰기 동작은 SIGPIPE를 생성할 것이다.

   c. non-blocking connect를 사용하는 socket는 연결이 종료되거나, connect를 실패 했을 때.

   d. socket error는 해결되지 못하고 있다. socket에서 쓰기 동작은 block되지 않을 것이고, 특정한 error 조건에서 설정된 errno으로 error (-1)을 리턴할 것이다. 이 pending error들은 getsockopt를 호출하고 SO_ERROR socket option을 제시함으로써 제거되거나 불려질 수 있다.

3. socket은 만약 socket을 위한 out-of-band data가 있거나 socket이 여전히 out-of-band 지점에 있다면 처리되지 않은 예외 조건을 가진다.

select에 의해서 readable과 writable 둘 다 기록되는 것처럼, socket에서 에러가 발생 했을 때 알린다. receive와 send 최저점의 목적은 select가 읽을 수 있는 혹은 쓸 수 있는 상태를 리턴하기 전에 얼마나 많은 data가 읽기를 위해 이용 가능한지 또는 얼마나 많은 공간이 쓰기를 위해 이용 가능한지를 조절하는 어플리케이션이 주어질 것이다.

Condition	Readable?	Writable?	Exception?
Data to read	•		
Read half of the connection closed	•		
New connection ready for listening socket	•		
Space available for writing		•	
Write half of the connection closed		•	
Pending error	•	•	
TCP out-of-band data			•

Figure 6.7 Summary of conditions that cause a socket to be ready for *select.*

Select를 위한 descriptor이 최대값

우리가 이전에 대부분의 어플리케이션이 많은 descriptor를 사용하지 않는다고 말했다. 예를 들면, 수 백 descriptor를 사용하는 어플리케이션을 찾기는 힘들다. 그러나, 그러한 어플리케이션은 분명 존재하고, 그리고 그것은 종종 다양한 descriptor에서 select를 사용한다. select가 처음 만들어질 때, OS는 일반적으로 process당 descriptor의 수에 제한이 있었고(4.2BSD의 제한은 31개), select 역시 이와 같은 제한이 있다. 그러나, Unix 현재 버전은 process당 descriptor의 수가 가상으로 무한대로 허용한다(종종 관리상의 제한, 메모리 양에 의해 제한이 존재한다), 그래서 질문은 다음과 같다: 이것은 select에 어떤 영향을 미치는가?

많은 구현에서는 다음과 같은 선언을 가진다, 그리고 그것은 4.4BSD <sys/types.h> header에서 가져왔다:

```
#ifndef     FD_SETSIZE
#define     FD_SETSIZE   256
#endif
```

이것은 우리를 우리가 단지 select에 의해서 사용될 descriptor 설정의 크기를 증가하기 위해서 이 header를 포함하기 전에 #define FD_SETSIZE가 몇몇 큰 값일 수도 있다는 것을 생각하도록 한다. 불행하게도, 일반적으로는 작동하지 않는다.

6.3 STR_CLI 함수: REVISITED

우리는 5장의 str_cli 함수를 다시 고쳐 쓸 수 있고, 이 번에는 select를 사용한다, 그래서 우리는 server process가 종료되자마자 알아차릴 수 있다. 초기 버전의 문제점은 socket에서 몇몇 경우 fgets를 호출할 때 block될 수 있다. 우리의 새로운 버전은 대신에 다른 표준 입력이나 읽을 수 있는 socket을 기다릴 때, select 호출을 block한다.

그림 6.8은 우리가 select를 호출함으로써 다뤄야 할 다양한 조건들을 보여준다.

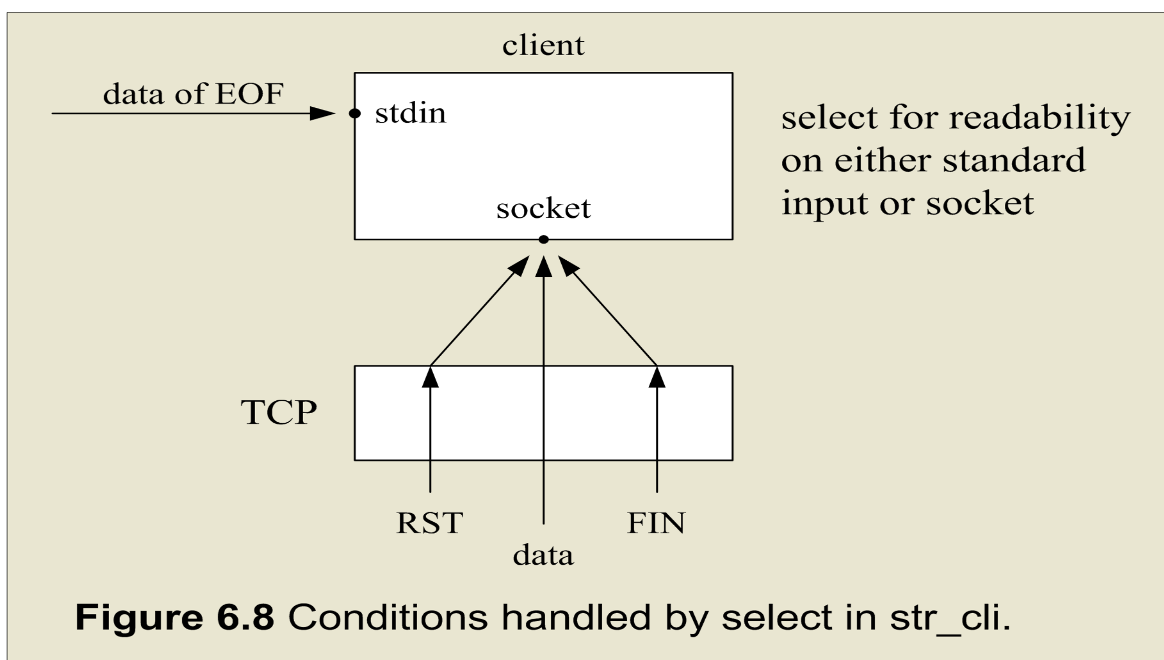


Figure 6.8 Conditions handled by select in str_cli.

세 가지 조건은 socket으로 다루어진다.

1. 만약 peer TCP가 data를 보낸다면, socket이 읽을 수 있어야 하고 read는 0보다 큰 값을 return한다(예를 들면, data의 byte 수).
2. 만약 peer TCP가 FIN을 보낸다면(peer process는 종료), socket이 읽을 수 있어야 하고 read는 0을 return한다(EOF).
3. 만약 peer TCP가 RST를 보낸다면(peer host는 충돌되어 재부팅), socket이 읽을 수 있어야 하고 read는 -1을 return하고, errno는 명확한 error 코드를 포함한다.

다음은 이러한 새로운 버전을 위한 소스코드를 보여준다.

```
<strcliselect01.c>
--------------------------------------------------------------------------
   1  #include             "lnp.h"
   2  #include             <unistd.h>
   3
   4  void
   5  str_cli(FILE *fp, int sockfd)
   6   {
   7          int    maxfdp1;
   8          fd_set rset;
   9          char   sendline[MAXLINE], recvline[MAXLINE];
   10         FD_ZERO(&rset);
   11         for ( ; ; ) {
   12                 FD_SET(fileno(fp), &rset);
   13                 FD_SET(sockfd, &rset);
   14                 maxfdp1 = max(fileno(fp), sockfd) + 1;
   15                 select(maxfdp1, &rset, NULL, NULL, NULL);
   16                 if (FD_ISSET(sockfd, &rset)) {/* socket is readable */
   17                         if (read (sockfd, recvline, MAXLINE) == 0)
   18                                 err_quit  ("str_cli:  server  terminated
      prematurely");
   19                         fputs (recvline, stdout);
   20                 }
   21                 if  (FD_ISSET  (fileno  (fp),  &rset)  )  {  /*  input  is
      readable */
   22                         if (fgets (sendline, MAXLINE, fp) == NULL)
   23                                 return;             /* all done */
   24                         write (sockfd, sendline, strlen(sendline) );
   25                 }
   26         }
   27 }
--------------------------------------------------------------------------
```

select 호출

우리는 단지 하나의 descriptor set이 필요하다-읽을 수 있는지를 체크하기 위해서. 이 set은 FD_ZERO에 의해서 초기화 되고 그러면 두 bit는 FD_SET을 사용하도록 작동시킨다: 표준 입/출력 파일 포인터, fp 와 일치하는 bit와 socket, sockfd와 일치하는 bit. select(그리고 poll)는 단지 descriptor와 함께 작동한다.

select는 두 descriptor의 최대값을 계산한 후에 호출된다. 호출에서, write-set 포인터와 exception-set 포인터 둘 다 null 포인터이다. 마지막 인자(시간 제한)은 역시 null 포인터이다. 왜냐하면 우리는 어떤 것이 준비 될 때까지 호출이 block되기를 원한다.

읽을 수 있는 socket 처리

select로부터 return에서, 만약 socket이 읽을 수 있다면, 에코된 줄은 readline으로 읽혀지고 fputs에 의해 출력된다.

쓸 수 있는 socket 처리

만약 표준 입력이 읽을 수 있다면, 한 줄은 fgets에 의해서 읽혀지고 writen을 사용하는 socket에서 쓰여진다.

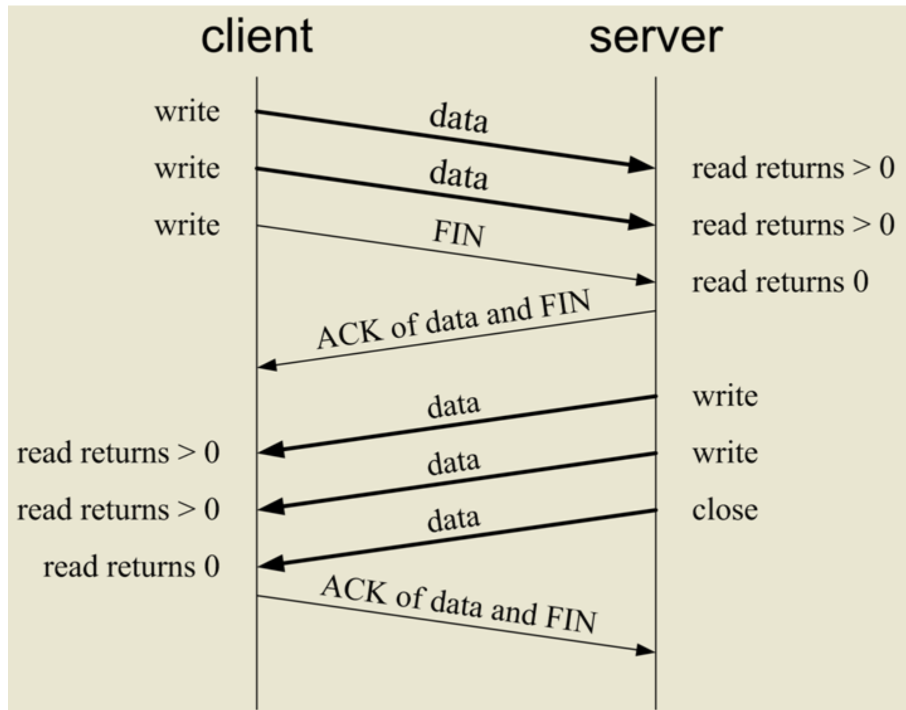
같은 네 개의 입/출력 함수로 fgets, writen, readline, 그리고 fputs가 사용된 것을 안다, 그러나 함수에서의 순서는 변했다. fgets를 호출에 의해 구동되는 함수 대신에, select를 현재 호출에 의해 구동된다. 예제에서 단지 몇 줄의 코드 추가로, 5절 예제와 비교해 보면, 우리가 우리의 클라이언트의 굉장한 부하를 추가했다.

6.4 SHUTDOWN 함수

네트워크 연결을 종료하는 일반적인 방법은 close함수를 호출하는 것이다. 그러나, shutdown으로 대체할 수 있는 close의 두 가지 제한이 있다.

1 close는 descriptor의 레퍼런스 수를 감소하고, 만약 count가 0에 도달할 경우 socket을 닫는다. 우리는 4장에서 이에 대해서 언급했다. shutdown으로, 우리는 레퍼런스 수에 관계없이, TCP의 일반적인 연결 종료 sequence를 초기화 할 수 있다.

2 close는 data 전송과 읽고 쓰기의 방향을 종료한다. TCP 연결은 full-duplex이기 때문에, 우리가 비록 우리에게 전송한 더 많은 data를 가질지라도, 우리가 전송을 종료했던 다른 전송을 알리기를 원하고 있다. 이것은 우리가 우리의 str_cli 함수에서 batch 입력으로 이전 section에서 언급했던 시나리오이다. 그림 6.9는 이 시나리오에서 전형적인 함수를 보여준다.



<그림 6.9> shutdown과 half-close

```
#include <sys/socket.h>

int shutdown (int sockfd, int howto);

                                        Returns: 0 if OK, -1 on error
```

함수의 작동은 howto 인자의 값에 의존한다

SHUT_RD 연결의 절반인 읽기는 닫혀있다-더 이상 data는 socket에서 receive되고 현재의 socket receive 버퍼에서 어떤 data는 버려진다. TCP socket을 위한 호출 후에 어떤 data는 인식되고 조용히 버려진다.

SHUT_WR 연결의 절반인 쓰기는 닫혀있다-TCP의 경우, 이것은 half-close라 불린다. 현재 socket send buffer에서 어떤 data는 전송될 것이고, TCP의 일반적인 전송 종료 sequence에 의해 뒤따르게 될 것이다.

SHUT_RDWR 연결의 절반인 읽기와 쓰기는 둘 다 종료된다-이것은 shutdown을 두 번 호출하는 것과 같다: 첫 번째는 SHUT_RD와 SHUT_WR.

6.5 STR_CLI 함수: REVISITED AGAIN

다음 코드는 개정된 str_cli 함수를 보여준다. 이 버전은 select와 shutdown을 사용한다.

<strcliselect02.c>

```
---------------------------------------------------------------------------
 1  #include            "lnp.h"
 2  #incldue            <unistd.h>
 3
 4  void
 5  str_cli(FILE *fp, int sockfd)
 6   {
 7          int    maxfdp1, stdineof;
 8          fd_set rset;
 9          char   buf[MAXLINE];
10          int    n;
11          stdineof = 0;
12          FD_ZERO(&rset);
13          for ( ; ; ) {
14                  if (stdineof == 0)
15                          FD_SET(fileno(fp), &rset);
16                  FD_SET(sockfd, &rset);
17                  maxfdp1 = max(fileno(fp), sockfd) + 1;
18                  select(maxfdp1, &rset, NULL, NULL, NULL);
19                  if (FD_ISSET(sockfd, &rset) ) {
20                          if ( (n = read (sockfd, buf, MAXLINE) == 0)
21                                  if (stdineof == 1)
22                                          return;
23                                  else
24                                     err_quit("str_cli:server terminated ");
25                          }
26                          write (fileno (stdout), buf, n);
27                  }
28                  if (FD_ISSET (fileno (fp), &rset) ) {
29                          if ( (n = read(fileno(fp), buf, MAXLINE) ) == 0)
            {
30                                  stdineof = 1;
31                                  shutdown (sockfd, SHUT_WR) ;/* FIN */
32                                  FD_CLR (fileno (fp), &rset);
33                                  continue;
34                          }
35                          write (sockfd, buf, n);
36                  }
37          }
38  }
```

stdineof는 0으로 초기화된 새로운 flag이다. 이 flag가 0인 동안, main loop를 도는 매 시간마다, 우리는 읽을 수 있기 위한 표준 입력에서 select한다.

우리가 socket에서 EOF를 읽을 때, 만약 우리가 이미 표준 입력에서 EOF를 만났다면, 이것은 일반적인 종료이고 함수는 return한다. 그러나 만약 우리가 아직 표준 입력에서 EOF를 만나지 않았다면, 서버 process는 조급하게 종료했다. 우리는 지금 read와 write가 line 대신에 버퍼를 작동하기 위해 호출하고 select가 예외적으로 작동하는 것을 허용한다.

6.6 TCP ECHO 서버: REVISITED

우리는 5절의 TCP echo 서버를 client당 하나의 child를 fork하는 대신에, 어떤 client의 수를 다루는 select를 사용하는 단일 프로세스로써 서버를 다시 쓸 수 있다. 다음 예제는 이 서버의 첫 번째 반을 보여준다.

<tcpcliserv/tcpservselect01.c>

```
------------------------------------------------------------------------
   1  #include            "lnp.h"
   2  int
   3  main (int argc, char **argv)
   4   {
   5         int    i, maxi, listenfd, connfd, sockfd;
   6         int    nready, client[FD_SETSIZE];
   7         ssize_t      n;
   8         fd_set rset, allset;
   9         char   buf[MAXLINE];
   10        socklen_t    clilen;
   11        struct sockaddr_in   cliaddr, servaddr;
   12        listenfd = Socket (AF_INET, SOCK_STREAM, 0);
   13        bzero (&servaddr, sizeof(servaddr));
   14        servaddr.sin_famliy = AF_INET;
   15        servaddr.sin_addr.s_addr = htonl (INADDR_ANY);
   16        servaddr.sin_port = htons (SERV_PORT);
   17        Bind (listenfd, (SA *) &servaddr, sizeof(servaddr));
   18        Listen (listenfd, LISTENQ);
   19        maxfd = listenfd;          /* initialize */
   20        maxi = -1;          /* index into client[] array */
   21        for (i = 0; i < FD_SETSIZE; i++)
   22              client[i] = -1;     /* -1 indicates available entry */
   23        FD_ZERO (&allset);
   24        FD_SET (listenfd, &allset);
```

listening socket을 만들고 select를 위한 초기화

listening socket을 창조하는 단계는 밖에서 보는 것과 같다: socket, bind, listen. 우리는 초기화할 listening socket에서 select할 descriptor를 가정하는 자료 구조를 초기화한다. 함수의 마지막 반이 다음 예제에서 보여진다.

```
<tcpservselect01.c>
   25        for ( ; ; ) {
   26                rset = allset;        /* structure assignment */
   27                nready = Select (maxfd + 1, &rset, NULL, NULL, NULL);
   28                if (FD_ISSET(listenfd, &rset)) {/* new client connection */
   29                        clilen = sizeof (cliaddr);
   30                        connfd = Accept (listenfd, (SA *) &cliaddr, &clilen);
   31                        for ( I = 0; I < FD_SETSIZE; i++) {
   32                                if (client[i] < 0) {
   33                                        client[i] = connfd;/*save descriptor*/
   34                                        break;
   35                                }
   36                        if (I == FD_SETSIZE)
   37                                err_quit ("too many clients");
   38                        FD_SET(connfd, &allset);/*add new descriptor to set*/
   39                        if ( connfd > maxfd)
   40                                maxfd = connfd;       /* for select */
   41                        if (i > maxi)
   42                                maxi = i;/* max index in client[] array */
   43                        if (--nready <= 0)
   44                                continue;/* no more readable descriptor */
   45                }
   46                for (i = 0; I <= maxi; i++) { /*check all clients for data */
   47                        if ( (sockfd = client[i] ) < 0)
   48                                continue;
   49                        if (FD_ISSET (sockfd, &rset) ) {
   50                                if ( (n = Read (sockfd, buf, MAXLINE)) == 0) {
   51                                        /* connection closed by client */
   52                                        Close (sockfd);
   53                                        FD_CLR (sockfd, &allset);
   54                                        client[i] = -1;
   55                                } else
   56                                        Writen (sockfd, buf, n);
   57                                if (--nready <= 0)
   58                                        break;  /*no more descriptors */
   59                        }
   60                }
   61        }
   62  }
```

----

select에서 Block

select는 일어날 무언가를 기다리고 있다: 새로운 client 연결의 성립 또는 data(FIN)의 도착, 또는 존재하는 연결에서 RST.

새로운 연결 accept

만약 listening socket가 읽을 수 있다면, 새로운 연결은 성립되었다. 우리는 accept를 호출하고 뒤를 이어 우리의 자료 구조를 업데이트 한다. 우리는 연결된 socket을 기록하는 client 배열에서 첫 번째 사용되지 않은 entry를 사용한다. ready descriptor의 수는 줄어들고, 만약 0이라면, 우리는 다음 for loop를 피할 수 있다. 이것은 준비 안된 descriptor를 체크를 피하기 위해 select로부터 return 값을 사용하도록 시킨다.

존재하는 연결 check

test는 하여간 그것의 descriptor가 select에 의해 return되는 descriptor set에서 있음으로써 각각의 존재하는 client 연결에 도움이 된다. 그래서 만약, 한 줄이 client로부터 읽혀진다면 client에 다시 echo된다. 만약 client 가 연결이 종료하면, read는 0을 return하고 우리는 이어서 우리의 자료 구조를 update한다.

DoS 공격

불행하게도, 방금 우리가 본 서버는 문제가 있다. 만약 악의 있는 client가 서버에 접속한다면, data(다른 newline)의 한 byte를 전송하고, sleep 상태가 지속하는 것을 발생하는 것을 간주한다. 서버는 read를 호출할 것이고, client로부터 data의 한 byte를 읽을 것이고 다음 호출에서 read를 block할 것이고, 이 client로부터 더 많은 data를 기다린다. 서버는 이 하나의 client에 의해 block되고("hung"이 더 좋은 표현이다), 악의적인 client는 newline을 전송하거나 종료할 때까지 다른 client에는 서비스 되지 않을 것이다(새로운 client 연결또는 존재하는 client의 data).

여기서 기본적인 개념은 서버가 다양한 client를 처리할 때, 서버는 결코 단일 client에 연관된 함수 호출에서 block될 수 없다. 그래서 서버를 정지할 수 있고 모든 다른 client에 서비스를 거부할 수 있다. 이것은 denial-of-service 공격으로 불려진다. 서버는 다른 합벅적인 client 서비스를 방해한다. 가능한 해결책은 (i) nonblocking I/O 사용, (ii) control을 분리한 thread에 의해 각각의 client에 서비스한다(예를 들면, 프로세스를 낳거나 각각의 client 서비스를 위한 thread), 또는 (iii) 입/출력에서 시간초과를 둔다.

6.7 PSELECT 함수

pselect함수는 POSIX에 의해 만들어졌고 현재 많은 Unix에 의해 지원된다.

```
#include <sys/select.h>

#include <signal.h>

#include <time.h>

int pselect (int maxfdp1, fd_set *readset, fd_set *writeset, fd_set
*exceptset, const struct timespec *timeout, const sigset_t *sigmask);

           Returns: count of ready descriptors, 0 on timeout, -1 on error
```

pselect는 일반적인 select 함수로부터 두 가지 변화를 포함한다.

1 pselect는 timespec 구조체를 사용한다, 다른 POSIX 창안, 대신에 timeval 구조체를 사용한다.

```
struct timespec {
       time_t tv_sec;              /* seconds */
       long   tv_nsec;             /* nanoseconds */
};
```

이 두 구조체에서 차이점은 두 번째 멤버이다: 새로운 구조체의 tv_nsec 멤버는 nanoseconds를 의미한다, 반면에 예전 구조체의 tv_usec 멤버는 microseconds를 의미한다.

2 pselect는 6번째 인자를 추가한다: signal mask 포인터. 이것은 특정한 signal의 전달을 억제하기 위한 프로그램을 호용하고, now-disabled signal을 위한 handler에 의해 설정된 몇몇 전역 변수를 테스트하고, pselect를 호출하고, signal mask를 초기화한다.

두 번째 관점에 주의하면, 다음 예제를 고려할 필요가 있다. SIGINT를 위한 우리의 프로그램의 signal handler는 단지 전역 intr_flag를 설정하고 return한다. 만약 우리의 process는 select를 호출에 block된다면, signal handler로부터의 return은 errno로 return한 함수로 하여금 EINTR로 설정하도록 한다. 그러나 select가 호출될 때, 코드는 다음과 같다:

```
if (intr_flag)
       handle_intr();             /* handle the signal */
if ( (nready = select ( … ) ) < 0) {
       if (errno == ENTER) {
              if (intr_flag)
                     handle_intr();
       }
       …
}
```

프로그램은 intr_flag의 test와 select 호출 사이에 있고, 만약 signal이 발생한다면, 그것은 select가 영원히 block하던지 아닌지 잊어버릴 것이다. pselect로, 우리는 지금 이 예제 코드를 신뢰할 수 있다.

```
sigset_t      newmask, oldmask, zeromask;

sigemptyset (&zeromask);

sigemptyset (&newmask);

sigaddset (&newmask, SIGINT) ;

sigprocmask (SIG_BLOCK, &newmask, &oldmask) ; /* block SIGIN */

if (intr_flag)

       handle_intr();      /* handle the signal */

if ( (nready = pselect ( … , &zeromask) ) < 0) {

       if (errno == EINTR) {

              if (intr_flag)

                     handle_intr();

       }

       …

}
```

intr_flag 변수를 test하기 전에, 우리는 SIGINT를 막는다. pselect가 호출될 때, 그것은 빈 set(예를 들면 zeromask)으로 process의 signal mask로 대체하고, descriptor를 check하고, 가능하면 sleep 한다. 그러나 pselect가 return할 때, process의 signal mask는 pselect가 호출되기 전 값을 초기화 한다(예를 들면, SIGINT가 block된다).

7. SOCKET OPTIONS

Socket에 영향을 미치는 옵션들을 파악하고 설정하는 데는 다양한 방법이 있다. 이 장에서는 setsockopt() 과 getsockopt() 함수를 다루는 것으로 시작해서, 다음으로 모든 옵션의 기본값을 출력하는 예제, 그리고 모든 socket 옵션을 자세히 설명할 것이다.

7.1 GETSOCKOPT 및 SETSOCKOPT 함수

```
#include <sys/socket.h>

int getsockopt(int sockfd, int level, int optname, void *optval, socklen_t *optlen);

int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen) ;
```

둘 다 OK에는 0을, 에러에는 -1을 반환한다

sockfd는 열린 socket descriptor를 참조해야만 한다. level은 옵션을 해석하는 시스템 내부의 코드를 지정한다: 기본 socket 코드 또는 protocol-specific 코드 (예: IPv4, IPv6, TCP, 또는 SCTP). optval 은 setsockopt이 옵션의 새 값을 어디에서 가져올지, 또는 getsockopt이 옵션의 현재 값을 어디에 저장할지를 가리키는 포인터이다. 이 변수의 크기는 setsockopt에서는 value 형식으로, 그리고 getsockopt에서는 value-result 형식으로 마지막 인자에 지정되어 있다.

그림 7.1과 7.2에는 getsockopt에 의해 질의되거나 또는 setsockopt에 의해 설정될 수 있는 옵션들이 정리되어 있다. "Datatype" 열은 각각의 옵션에 대해 optval 포인터가 가리켜야 하는 datatype을 보여준다. 우리는 structure를 표시할 때 두 개의 brace를 사용해서 표시할 것이다, 예를 들어 linger{}는 struct linger를 의미한다.

옵션에는 두 개의 기본적인 타입이 있다: 특정 기능을 활성화시키거나 비활성화시키는 이진 옵션(flags), 그리고 우리가 설정하거나 살펴볼 수 있는 특정 값들을 입력하거나 가져오는 옵션(values). "Flag" 로 표시된 열은 옵션이 flag 옵션인지를 표시한다. 이런 flag 옵션들로 getsockopt를 호출할 때, *optval은 integer이다. *optval로 반환되는 값은 옵션이 비활성화되어 있을 경우 0이고, 옵션이 활성화되어 있으면 0이 아닌 값이다. 비슷하게, setsockopt은 옵션을 켜기 위해 0이 아닌 값을, 그리고 옵션을 끄기 위해 0의 값을 가진 *optval을 필요로 한다. "Flag" 열에 "●"가 없으면, 그 옵션은 user process와 system사이에 지정된 datatype의 값을 전달하는 데 사용된다.

level	optname	get	set	Description	Flag	Datatype
SOL_SOCKET	SO_BROADCAST	•	•	Permit sending of broadcast datagrams	•	int
	SO_DEBUG	•	•	Enable debug tracing	•	int
	SO_DONTROUTE	•	•	Bypass routing table lookup	•	int
	SO_ERROR	•		Get pending error and clear		int
	SO_KEEPALIVE	•	•	Periodically test if connection still alive	•	int
	SO_LINGER	•	•	Linger on close if data to send		linger{}
	SO_OOBINLINE	•	•	Leave received out-of-band data inline	•	int
	SO_RCVBUF	•	•	Receive buffer size		int
	SO_SNDBUF	•	•	Send buffer size		int
	SO_RCVLOWAT	•	•	Receive buffer low-water mark		int
	SO_SNDLOWAT	•	•	Send buffer low-water mark		int
	SO_RCVTIMEO	•	•	Receive timeout		timeval{}
	SO_SNDTIMEO	•	•	Send timeout		timeval{}
	SO_REUSEADDR	•	•	Allow local address reuse	•	int
	SO_REUSEPORT	•	•	Allow local port reuse	•	int
	SO_TYPE	•		Get socket type		int
	SO_USELOOPBACK	•	•	Routing socket gets copy of what it sends	•	int
IPPROTO_IP	IP_HDRINCL	•	•	IP header included with data	•	int
	IP_OPTIONS	•	•	IP header options		(see text)
	IP_RECVDSTADDR	•	•	Return destination IP address	•	int
	IP_RECVIF	•	•	Return received interface index	•	int
	IP_TOS	•	•	Type-of-service and precedence		int
	IP_TTL	•	•	TTL		int
	IP_MULTICAST_IF	•	•	Specify outgoing interface		in_addr{}
	IP_MULTICAST_TTL	•	•	Specify outgoing TTL		u_char
	IP_MULTICAST_LOOP	•	•	Specify loopback		u_char
	IP_ADD_MEMBERSHIP		•	Join multicast group		ip_mreq{}
	IP_DROP_MEMBERSHIP		•	Leave multicast group		ip_mreq{}
	IP_BLOCK_SOURCE		•	Block multicast source		ip_mreq_source{}
	IP_UNBLOCK_SOURCE		•	Unblock multicast source		ip_mreq_source{}
	IP_ADD_SOURCE_MEMBERSHIP		•	Join source-specific multicast		ip_mreq_source{}
	IP_DROP_SOURCE_MEMBERSHIP		•	Leave source-specific multicast		ip_mreq_source{}
IPPROTO_ICMPV6	ICMP6_FILTER	•	•	Specify ICMPv6 message types to pass		icmp6_filter{}
IPPROTO_IPV6	IPV6_CHECKSUM	•	•	Offset of checksum field for raw sockets		int
	IPV6_DONTFRAG	•	•	Drop instead of fragment large packets	•	int
	IPV6_NEXTHOP	•	•	Specify next-hop address		sockaddr_in6{}
	IPV6_PATHMTU	•		Retrieve current path MTU		int
	IPV6_RECVDSTOPTS	•	•	Receive destination options	•	int
	IPV6_RECVHOPLIMIT	•	•	Receive unicast hop limit	•	int
	IPV6_RECVHOPOPTS	•	•	Receive hop-by-hop options	•	int
	IPV6_RECVPATHMTU	•	•	Receive path MTU	•	int
	IPV6_RECVPKTINFO	•	•	Receive packet information	•	int
	IPV6_RECVRTHDR	•	•	Receive source route	•	int
	IPV6_RECVTCLASS	•	•	Receive traffic class	•	int
	IPV6_UNICAST_HOPS	•	•	Default unicast hop limit		int
	IPV6_USE_MIN_MTU	•	•	Use minimum MTU	•	int
	IPV6_V6ONLY	•	•	Disable v4 compatibility	•	int
	IPV6_XXX	•	•	Sticky ancillary data		(see text)
	IPV6_MULTICAST_IF	•	•	Specify outgoing interface		u_int
	IPV6_MULTICAST_HOPS	•	•	Specify outgoing hop limit		int
	IPV6_MULTICAST_LOOP	•	•	Specify loopback	•	u_int
	IPV6_JOIN_GROUP		•	Join multicast group		ipv6_mreq{}
	IPV6_LEAVE_GROUP		•	Leave multicast group		ipv6_mreq{}
IPPROTO_IP or IPPROTO_IPV6	MCAST_JOIN_GROUP		•	Join multicast group		group_req{}
	MCAST_LEAVE_GROUP		•	Leave multicast group		group_source_req{}
	MCAST_BLOCK_SOURCE		•	Block multicast source		group_source_req{}
	MCAST_UNBLOCK_SOURCE		•	Unblock multicast source		group_source_req{}
	MCAST_JOIN_SOURCE_GROUP		•	Join source-specific multicast		group_source_req{}
	MCAST_LEAVE_SOURCE_GROUP		•	Leave source-specific multicast		group_source_req{}

Figure 7.1

Summary of socket and IP-layer socket options for getsockopt and setsockopt

level	optname	get	set	Description	Flag	Datatype
IPPROTO_TCP	TCP_MAXSEG	•	•	TCP maximum segment size		int
	TCP_NODELAY	•	•	Disable Nagle algorithm	•	int
IPPROTO_SCTP	SCTP_ADAPTION_LAYER	•	•	Adaption layer indication		sctp_setadaption{}
	SCTP_ASSOCINFO	†	•	Examine and set association info		sctp_assocparams{}
	SCTP_AUTOCLOSE	•	•	Autoclose operation		int
	SCTP_DEFAULT_SEND_PARAM	•	•	Default send parameters		sctp_sndrcvinfo{}
	SCTP_DISABLE_FRAGMENTS	•	•	SCTP fragmentation	•	int
	SCTP_EVENTS	•	•	Notification events of interest		sctp_event_subscribe{}
	SCTP_GET_PEER_ADDR_INFO	†		Retrieve peer address status		sctp_paddrinfo{}
	SCTP_I_WANT_MAPPED_V4_ADDR	•	•	Mapped v4 address	•	int
	SCTP_INITMSG	•	•	Default INIT parameters		sctp_initmsg{}
	SCTP_MAXBURST	•	•	Maximum burst size		int
	SCTP_MAXSEG	•	•	Maximum fragmentation size		int
	SCTP_NODELAY	•	•	Disable Nagle algorithm	•	int
	SCTP_PEER_ADDR_PARAMS	†	•	Peer address parameters		sctp_paddrparams{}
	SCTP_PRIMARY_ADDR	†	•	Primary destination address		sctp_setprim{}
	SCTP_RTOINFO	†	•	RTO information		sctp_rtoinfo{}
	SCTP_SET_PEER_PRIMARY_ADDR		•	Peer primary destination address		sctp_setpeerprim{}
	SCTP_STATUS	†		Get association status		sctp_status{}

Figure 7.2 Summary of transport-layer socket options.

7.2 SOCKET OPTION 상태 검사하기

우리는 이제 그림 7.1과 7.2에 정의되어 있는 대부분의 옵션들이 지원되는지 확인하고, 지원된다면, 옵션들의 기본값을 출력하는 프로그램을 작성할 것이다. 그림 7.3에 우리 프로그램을 위한 선언들이 있다.

가능한 값들의 union 선언하기

3-8 우리의 union은 getsockopt에서 반환할 수 있는 가능한 각각의 반환값들을 위한 하나의 멤버를 포함하고 있다.

함수 prototype 정의하기

9-12 우리는 주어진 socket 옵션에 대해 값을 출력하기 위해 호출된 네 개의 함수를 위한 함수 prototype을 정의한다.

Structure를 정의하고 array를 초기화하기

13-52 우리의 sock_opts structure는 각각의 socket 옵션으로 getsockopt를 호출하는 데 필요한 모든 정보들을 포함하고 있고 그 정보들의 현재 값을 반환한다. 마지막 멤버, opt_val_str은, 옵션 값을 출력할 우리의 네 함수들 중 하나로의 pointer이다. 우리는 각 socket 옵션의 인자들에 대해, 이런 structure의 array를 할당하고 초기화한다.

sockopt/checkopts.c

```
 1 #include    "unp.h"
 2 #include    <netinet/tcp.h>     /* for TCP_xxx defines */

 3 union val {
 4   int               i_val;
 5   long              l_val;
 6   char              c_val[10];
 7   struct linger     linger_val;
 8   struct timeval    timeval_val;
 9 } val;

10 static char *sock_str_flag(union val *, int);
11 static char *sock_str_int(union val *, int);
12 static char *sock_str_linger(union val *, int);
13 static char *sock_str_timeval(union val *, int);

14 struct sock_opts {
15   char     *opt_str;
16   int       opt_level;
17   int       opt_name;
18   char    *(*opt_val_str)(union val *, int);
19 } sock_opts[] = {
20     "SO_BROADCAST",     SOL_SOCKET, SO_BROADCAST,   sock_str_flag,
21     "SO_DEBUG",         SOL_SOCKET, SO_DEBUG,       sock_str_flag,
22     "SO_DONTROUTE",     SOL_SOCKET, SO_DONTROUTE,   sock_str_flag,
23     "SO_ERROR",         SOL_SOCKET, SO_ERROR,       sock_str_int,
24     "SO_KEEPALIVE",     SOL_SOCKET, SO_KEEPALIVE,   sock_str_flag,
25     "SO_LINGER",        SOL_SOCKET, SO_LINGER,      sock_str_linger,
26     "SO_OOBINLINE",     SOL_SOCKET, SO_OOBINLINE,   sock_str_flag,
27     "SO_RCVBUF",        SOL_SOCKET, SO_RCVBUF,      sock_str_int,
28     "SO_SNDBUF",        SOL_SOCKET, SO_SNDBUF,      sock_str_int,
29     "SO_RCVLOWAT",      SOL_SOCKET, SO_RCVLOWAT,    sock_str_int,
30     "SO_SNDLOWAT",      SOL_SOCKET, SO_SNDLOWAT,    sock_str_int,
31     "SO_RCVTIMEO",      SOL_SOCKET, SO_RCVTIMEO,    sock_str_timeval,
32     "SO_SNDTIMEO",      SOL_SOCKET, SO_SNDTIMEO,    sock_str_timeval,
33     "SO_REUSEADDR",     SOL_SOCKET, SO_REUSEADDR,   sock_str_flag,
34 #ifdef  SO_REUSEPORT
35     "SO_REUSEPORT",     SOL_SOCKET, SO_REUSEPORT,   sock_str_flag,
36 #else
37     "SO_REUSEPORT",     0,          0,              NULL,
38 #endif
39     "SO_TYPE",          SOL_SOCKET, SO_TYPE,        sock_str_int,
40     "SO_USELOOPBACK",   SOL_SOCKET, SO_USELOOPBACK, sock_str_flag,
41     "IP_TOS",           IPPROTO_IP, IP_TOS,         sock_str_int,
42     "IP_TTL",           IPPROTO_IP, IP_TTL,         sock_str_int,
43     "TCP_MAXSEG",       IPPROTO_TCP,TCP_MAXSEG,     sock_str_int,
44     "TCP_NODELAY",      IPPROTO_TCP,TCP_NODELAY,    sock_str_flag,
45     NULL,               0,          0,              NULL
46 };
```

sockopt/checkopts.c

<그림 7.3> socket 옵션들을 확인하기 위한 우리 program에서의 선언들

그림 7.4는 main 함수를 보여준다.

```
                                                              sockopt/checkopts.c
47 int
48 main(int argc, char **argv)
49 {
50     int     fd, len;
51     struct sock_opts *ptr;

52     fd = Socket(AF_INET, SOCK_STREAM, 0);

53     for (ptr = sock_opts; ptr->opt_str != NULL; ptr++) {
54         printf("%s: ", ptr->opt_str);
55         if (ptr->opt_val_str == NULL)
56             printf("(undefined)\n");
57         else {
58             len = sizeof(val);
59             if (getsockopt(fd, ptr->opt_level, ptr->opt_name,
60                            &val, &len) == -1) {
61                 err_ret("getsockopt error");
62             } else {
63                 printf("default = %s\n", (*ptr->opt_val_str) (&val, len));
64             }
65         }
66     }
67     exit(0);
68 }
                                                              sockopt/checkopts.c
```

<그림 7.4> main 함수에서 모든 socket 옵션들을 확인

모든 옵션들을 다 둘러보기

59-63 우리는 array 안에 있는 모든 인자들을 다 둘러본다. 만약 opt_val_str 포인터가 null이면, 그 옵션은 implementation에 의해 정의되지 않는다 (우리가 SO_REUSEPORT에서 보여주었다).

Socket 생성하기

63-82 우리는 옵션을 시험해 볼 socket을 생성한다. socket, IPv4, 그리고 TCP layer socket 옵션을 시험해보기 위해, 우리는 IPv4 TCP socket을 사용한다. IPv6 layer socket 옵션을 사용하기 위해, 우리는 IPv6 TCP socket을 사용하고, SCTP layer socket 옵션들을 시험하기 위해, 우리는 IPv4 SCTP socket을 사용한다.

getsockopt 호출하기

83-87 우리는 getsockopt을 호출하지만 에러가 반환될 시 종료하지 않는다. 많은 구현들이 socket 옵션들 중 일부를 그들이 옵션을 지원하지 않더라도 정의한다. 지원되지 않는 옵션들은 ENOPROTOOPT 에러를 유도해 낼 것이다.

옵션의 기본값 출력하기

88-89 만약 getsockopt가 성공적으로 리턴하면, 우리는 옵션 값을 문자열로 변환하고 그 문자열을 출력하기 위해 우리의 함수를 호출한다.

그림 7.3에서, 우리는 각각의 옵션 값들이 반환되는, 네 개의 함수 prototype을 보여주었다. 그림 7.5는 이런 네 함수들 중 하나인, flag 옵션의 값을 출력하는, sock_str_flag 함수를 보여준다. 다른 세 함수도 비슷하다.

```
                                                        sockopt/checkopts.c
69 static char strres[128];

70 static char *
71 sock_str_flag(union val *ptr, int len)
72 {
73     if (len != sizeof(int))
74         snprintf(strres, sizeof(strres), "size (%d) not sizeof(int)", len);
75     else
76         snprintf(strres, sizeof(strres),
77                  "%s", (ptr->i_val == 0) ? "off" : "on");
78     return(strres);

79 }
                                                        sockopt/checkopts.c
```

<그림 7.5> sock_str_flag 함수: flag 옵션을 문자열로 변환

99-104 getsockopt의 마지막 argument가 value-result argument라는 것을 기억하라. 우리가 확인해볼 첫 번째 사항은 getsockopt에 의해 반환된 값의 크기가 예상한 크기인지이다. 반환된 문자열은 flag 옵션의 값이 0인지 0이 아닌 값인지에 따라 각각 off 또는 on이다.

리눅스에서 이 프로그램을 실행하면 다음의 출력이 나온다:

```
root@localhost:~/LNP/unp/sockopt
[root@localhost sockopt]# ./checkopts
SO_BROADCAST: default = off
SO_DEBUG: default = off
SO_DONTROUTE: default = off
SO_ERROR: default = 0
SO_KEEPALIVE: default = off
SO_LINGER: default = l_onoff = 0, l_linger = 0
SO_OOBINLINE: default = off
SO_RCVBUF: default = 87380
SO_SNDBUF: default = 16384
SO_RCVLOWAT: default = 1
SO_SNDLOWAT: default = 1
SO_RCVTIMEO: default = 0 sec, 0 usec
SO_SNDTIMEO: default = 0 sec, 0 usec
SO_REUSEADDR: default = off
SO_REUSEPORT: (undefined)
SO_TYPE: default = 1
SO_USELOOPBACK: (undefined)
IP_TOS: default = 0
IP_TTL: default = 64
IPV6_DONTFRAG: (undefined)
IPV6_UNICAST_HOPS: Can't create fd for level 41

[root@localhost sockopt]#

[영어 ][완성 ][두벌식 ]
```

Socket States

몇몇 socket 옵션들은 언제 옵션들을 설정하거나 가져올 지와 socket 의 상태에 대한 시간과 관계된 고려사항을 가지고 있다. 우리는 영향 받는 옵션들과 함께 이것을 언급한다.

다음의 socket 옵션들은 listen중인 socket에서 연결된 TCP socket에 의해 상속된다: SO_DEBUG, SO_DONTROUTE, SO_KEEPALIVE, SO_LINGER, SO_OOBINLINE, SO_RCVBUF, SO_RCVLOWAT, SO_SNDBUF, SO_SNDLOWAT, TCP_MAXSEG, 그리고 TCP_NODELAY.

TCP 계층에 의해 three-way handshake가 끝나기 전까지는 연결된 socket이 accept에 의해 서버로 반환되지 않기 때문에 이것은 TCP에 중요하다. three-way handshake가 완료될 때 연결된 socket에 이런 socket 옵션들이 설정되어 있다는 것을 보증하기 위해, 우리는 listen중인 socket에 그 옵션을 설정해야만 한다.

7.3 GENERIC SOCKET OPTIONS

우리는 generic socket 옵션에 관한 논의를 시작할 것이다. 이 옵션들은 protocol-independent(이 말은, 이 옵션들이 IPv4같은 하나의 특정 protocol 모듈에 의해서가 아니라, kernel 내부의 protocol-independent 코드에 의해 다뤄진다는 뜻이다) 이다, 그렇지만 옵션들 중 일부는 특정 형태의 socket들에만 적용된다. 예를 들어 SO_BROADCAST socket 옵션이 "generic"으로 불리더라도 이 옵션은 datagram socket들에만 적용된다.

SO_BROADCAST Socket Option

이 옵션은 broadcast message를 송신하는 process의 기능을 활성화시키거나 비활성화시킨다. broadcasting은 datagram socket 용으로만 그리고 broadcast message의 개념을 지원하는 망에서만 지원된다. 여러분은 point-to-point link 또는 SCTP 또는 TCP같은 connection-based transport protocol에서는 broadcast를 사용할 수 없다.

application이 broadcast datagram을 송신하기 전에 이 socket 옵션을 설정해야만 하는 관계로, 이 옵션은 application이 broadcast용으로 설계되지 않았을 경우 process가 broadcast를 실행하는 것을 제한한다. 예를 들어, 어떤 UDP application은 목적지 IP 주소를 command-line 인자로 받을 수도 있지만, 이 application은 사용자가 broadcast 주소를 입력하도록 의도된 것이 아니다. application이 주어진 주소가 broadcast 주소인지 아닌지를 판단하게 강요하는 대신에, 테스트는 kernel 내부에 있다: 목적지 주소가 broadcast 주소이고 이 socket 옵션이 설정되어 있지 않으면, EACCES 가 반환된다.

SO_DEBUG Socket Option

이 옵션은 TCP에 의해서만 지원된다. TCP socket 용으로 활성화되었을 때, kernel은 그 socket 용으로 TCP에 의해 송신되거나 수신된 모든 패킷에 대한 자세한 정보를 얻는다. 이 정보들은 trpt 프로그램으로 조사할 수 있는 kernel 내부의 circular buffer에 저장된다.

SO_DONTROUTE Socket Option

이 옵션은 나가는 패킷들이 하부 protocol의 normal routing mechanism들을 우회할 것을 지시한다. 예를 들어, IPv4를 사용할 때, 패킷은 목적지 주소의 network과 subnet 부분에 기술된 대로, 적절한 local interface쪽으로 유도된다. 만약 local interface가 목적지 주소에서 결정되지 않으면 (e.g 목적지가 point-to-point link 의 다른 end 에 있지 않거나, 또는 공유된 망에 있지 않을 때), ENETUNREACH가 반환된다.

이 옵션과 동일한 것이 또한 send, sendto, 또는 sendmsg 함수들에서 MSG_DONTROUTE flag를 사용해서 각각의 datagram들에 적용될 수 있다. 이 옵션은 보통 routing daemon들이 routing table을 우회하고 패킷을 특정 interface로 보내도록 강제하는 데 사용된다.

SO_ERROR Socket Option

어떤 socket에서 에러가 발생할 때, kernel 내부의 protocol module은 그 socket용으로 so_error라는 변수를 standard Unix Exxx 값들 중 하나로 설정한다. 이것을 socket을 위한 pending error라고 부른다. process는 하나 또는 두 가지의 방법으로 에러를 즉시 통보 받을 수 있다:

1) 만약 process가 socket에서 select를 호출하는 중 block되었을 때, readability나 writability 둘 다를 위해서, select는 두 개의 조건이 설정된 상태에서 반환된다.

2) 만약 process가 signal-driven I/O를 사용하면, SIGIO signal이 process 또는 process group 둘 다를 위해서 생성된다.

process는 그 뒤 SO_ERROR socket 옵션을 가져옴으로써 so_error의 값을 얻을 수 있다. getsockopt에 의해 반환되는 정수값은 socket을 위한 pending error이다. 그 뒤 so_error의 값은 kernel에 의해 다시 0으로 설정된다.

만약 process가 read를 호출하고 반환할 데이터가 없을 때 so_error가 0이 아닌 값이면, read는 errno를 so_error의 값으로 설정하고 -1을 반환한다. 그 후 so_error의 값은 0으로 재설정된다. 만약 그 socket에 queue된 데이터가 있으면, error condition 대신 그 데이터가 read로 반환된다. 만약 process가 write를 호출할 때 so_error가 0이 아닌 값이면, errno를 so_error의 값으로 설정하고 -1이 반환된 뒤 so_error는 0으로 재설정된다.

이 옵션은 우리가 가져올 수는 있지만 설정할 수 없는 옵션이다.

SO_KEEPALIVE Socket Option

어떤 TCP socket 용으로 keep-alive 옵션이 설정되어 있고 두 시간 동안 이 socket을 통해서 양쪽 방향으로 어떤 데이터도 교환되지 않았다면, TCP는 자동으로 peer에게 keep-alive probe를 전송한다. 이 probe는 peer가 반드시 응답해야 하는 TCP segment이다.

다음 3가지 시나리오 중 하나로 결론지어진다:

1. peer가 예상한 ACK로 응답하는 경우. application에게는 통보하지 않는다(모든 것이 정상이므로). 다음에 2시간 동안 활동이 없을 경우 TCP는 다른 probe를 전송할 것이다.

2. peer가 RST로 응답하는 경우, local TCP에게 peer host가 crash되었거나 reboot되었다는 것을 알려주는 것이다. socket의 pending error 는 ECONNRESET으로 설정되고 socket은 close된다.

3. keep-alive probe에 대해 peer로부터 응답이 없는 경우. Berkeley-derived TCP는 응답을 유도해내기 위해, 75초 간격으로, 8개의 추가적인 probe들을 전송한다. TCP는 첫 번째 probe를 전송한 뒤 11분 15초 사이에 응답이 오지 않을 경우 수신을 포기한다.

   만약 TCP의 keep-alive probe 전부에 대해 응답이 없을 경우, socket의 pending error는 ETIMEOUT으로 설정되고 socket 은 close된다. 그렇지만 만약 socket이 keep-alive probe들 중 하나에 대한 응답으로 ICMP error를 받을 경우, 다음의 error가 대신 반환된다 (그리고 socket은 close된다). 이 시나리오에서 일반적인 ICMP error는 "host unreachable"로, peer host가 unreachable 하다는 것을 나타내고, 이 경우 pending error는 EHOSTUNREACH로 설정된다. network failure 또는 remote host 가 crash되고 last-hop router가 이 crash를 감지했을 경우 모두 이 에러가 발생된다.

이 옵션의 용도는 peer host가 crash되었는지 또는 unreachable 상태()가 되었는지를 탐지하는 것이다 (e.g., dial-up modem connection drops, power fails, etc.). 만약 peer process가 crash되면, process의 TCP는 연결을 통해 FIN 을 송신할 것이고, 우리가 select로 쉽게 탐지할 수 있다. 또한 이해해야할 것은 어떤 keep-alive probe들에 대해서도 응답이 없다면(scenario 3), 우리는 peer host가 crash되었다는 것을 보장할 수 없고, TCP는 유효한 연결을 종료할 수도 있다. 몇몇 intermediate router가 15분 동안 crash되어서, 그 시간 간격이 우연하게 우리 host의 keep-alive probe 간격인 11분 15초를 완전히 넘어섰을 수도 있다. 사실 이 기능은 정상적인 연결들을 종료시킬 수 있기 때문에 "keep-alive"라고 부르기보다 "make-dead"라고 부르는 것이 더 적절할 수도 있다.

이 옵션은 클라이언트에서도 사용할 수 있지만, 보통 서버에서 사용된다. 서버들은 TCP 연결을 통해서 입력이 들어오는 것, 다시 말해서, client request를 기다리는 데 대부분의 시간을 보내기 때문에 이 옵션을 사용한다. 그렇지만 만약 클라이언트 host의 연결이 drop되거나, power off이거나, 또는 crash되면, 서버 process는 그 상황에 대해 절대로 알지 못할 것이고, 서버는 절대로 올 수 없는 입력을 계속 기다릴 것이다. 이것을 half-open connection이라 부른다. keep-alive 옵션은 이런 half-open connection들을 탐지해서 종료시킬 것이다.

몇몇 서버들, 특히 FTP 서버들은, 보통 몇 분 단위로, application timeout을 제공한다. 이 timeout의 application 내부의, 보통 read 호출 근처에서, 클라이언트의 다음 명령을 읽으면서 이루어진다. 이 timeout은 이 socket 옵션을 포함하지 않는다. application이 자체적으로 timeout을 구현했다면 이것을 완전히 통제할 수 있기 때문에, 이 방법은 사라진 클라이언트와의 연결을 제거하는 데 일반적으로 더 좋은 방법이다.

그림 7.6에는 TCP 연결의 다른 쪽 end에 어떤 일이 생기면 우리가 알아야 할 다양한 방법들이 요약되어 있다. 우리가 "using select for readability"라고 말할 때는, 어떤 socket이 readable한지를 확인하기 위해 select를 호출한다는 의미이다.

Scenario	Peer process crashes	Peer host crashes	Peer host is unreachable
Our TCP is actively sending data	Peer TCP sends a FIN, which we can detect immediately using `select` for readability. If TCP sends another segment, peer TCP responds with RST. If TCP sends yet another segment, our TCP sends us `SIGPIPE`.	Our TCP will time out and our socket's pending error is set to `ETIMEDOUT`.	Our TCP will time out and our socket's pending error is set to EHOSTUNREACH.
Our TCP is actively receiving data	Peer TCP will send a FIN, which we will read as a (possibly premature) end-of-file.	We will stop receiving data.	We will stop receiving data.
Connection is idle, keepalive set	Peer TCP sends a FIN, which we can detect immediately using `select` for readability.	Nine keepalive probes are sent after 2 hours of inactivity and then our socket's pending error is set to `ETIMEDOUT`.	Nine keepalive probes are sent after 2 hours of inactivity and then our socket's pending error is set to EHOSTUNREACH.
Connection is idle, keepalive not set	Peer TCP sends a FIN, which we can detect immediately using `select` for readability.	(Nothing.)	(Nothing.)

<그림 7.6> 다양한 TCP 조건들을 확인하기 위한 방법들

SO_LINGER Socket Option

이 옵션은 connection-oriented protocol을 위해 close함수가 어떻게 동작할지를 지정한다 (e.g., TCP 와 SCTP를 위해, UDP는 아님). 기본적으로, close는 즉시 리턴하지만, 만약 socket send buffer 내부에 어떤 데이터라도 계속 남아있다면, system은 그 data를 peer에게 전달할 것이다.

SO_LINGER socket 옵션은 우리가 이 기본값을 변경할 수 있게 해 준다. 이 옵션은 user process 와 kernel사이에 전달되는 다음의 structure를 필요로 한다. 이 structure는 <sys/socket.h>를 include함으로써 정의된다.

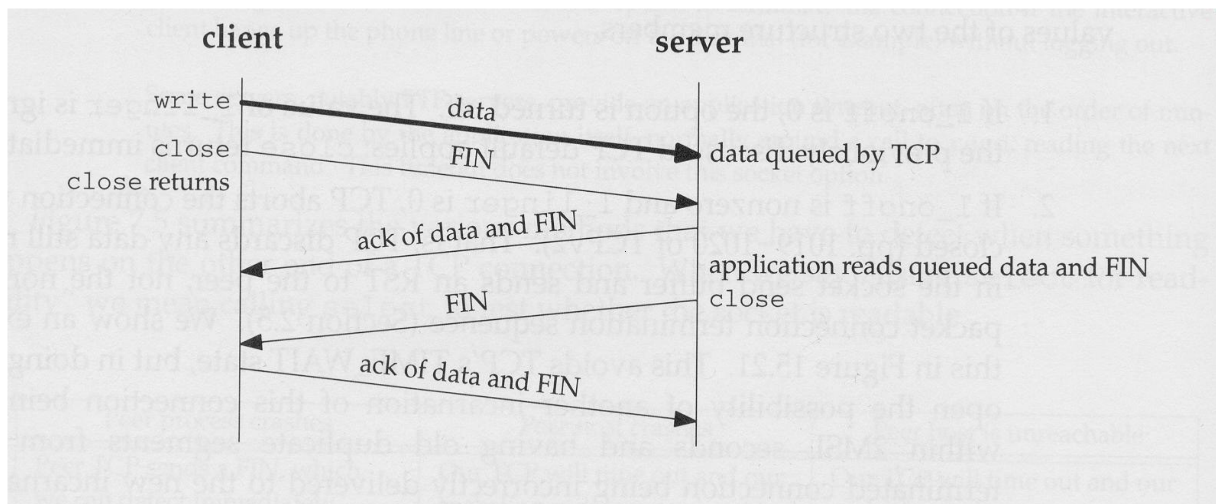
```
struct linger{
  int   l_onoff;    /* 0=off, nonzero=on */
  int   l_linger;   /* linger time, POSIX specifies units as seconds */
};
```

Setsockopt를 호출하면 두 개의 structure member들의 값에 따라서, 다음 세 가지 시나리오 중 하나로 이어진다:

1. 만약 l_onoff가 0이면, 옵션은 꺼진다. l_linger의 값은 무시되고 이전에 논의했던 TCP의 기본 동작이 적용된다: close은 즉시 리턴된다.
2. l_onoff가 0이 아닌 값이고 l_linger가 0이면, TCP는 연결이 close될 때 연결을 abort한다. 이 말은, TCP는 socket send buffer에 아직 남아있는 data를 버리고 peer에게 일반적인 four-packet connection termination sequence 대신, RST를 송신할 것이다. RST를 송신하는 것으로 TCP의 TIME_WAIT state를 피할 수 있지만, 이렇게 함으로써, 2MSL 초 안에 이 연결이 다시 생성되고 방금 종료된 연결에 있던 이전의 중복 segment들이 새로 부활된 연결로 잘못 전송될 가능성이 있다.
3. 만약 l_onoff가 0이 아닌 값이고 l_linger가 0이 아닌 값이면, socket이 close될 때 kernel은 남아있을 것이다. 이 말은, socket send buffer에 어떤 데이터라도 아직 남아있다면, process는 다음의 상황이 발생할 때까지 sleep 상태로 될 것이다: (i)모든 데이터가 송신되고 peer TCP에 의해 확인되었을 경우, 또는 (ii) linger time이 만기되었을 경우. 만약 socket이 nonblocking으로 설정되어 있었다면, process는 linger time이 0이 아니더라도,

close가 완료되기를 기다리지 않을 것이다. 만약 남은 data가 송신되고 확인 받기 전에 linger time이 만기되면, close는 EWOULDBLOCK을 반환하고 send buffer 내부에 남은 어떤 데이터라도 버려지게 되기 때문에, SO_LINGER 옵션에서 이 기능을 사용할 때, application이 close의 반환값을 확인하게 하는 것이 중요하다.

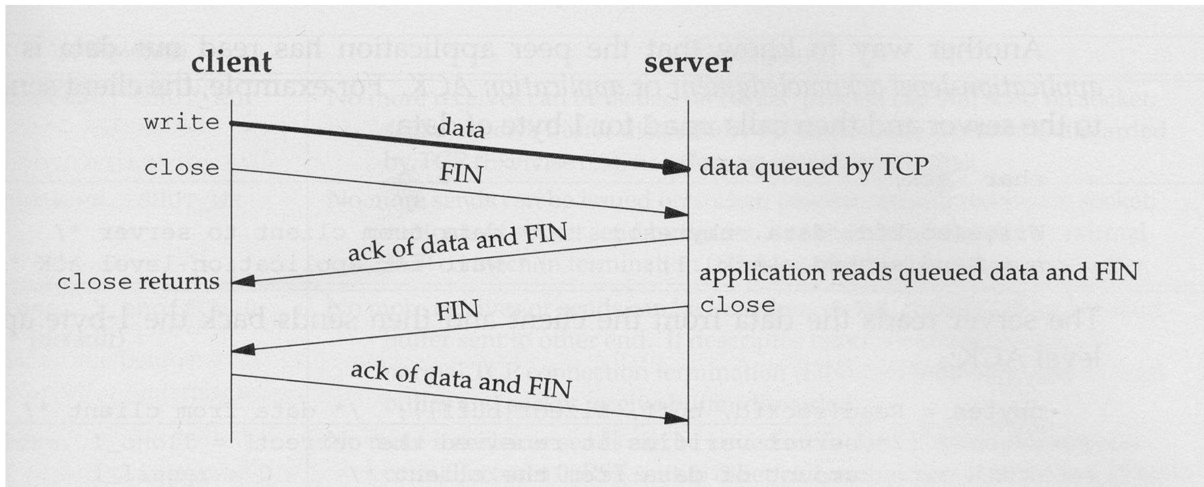
우리는 이제 우리가 살펴본 다양한 시나리오로부터 socket에서 close가 정확하게 언제 리턴되는지를 볼 필요가 있다. 우리는 클라이언트가 socket으로 data를 write하고 close를 호출한다고 가정한다. 그림 7.7은 기본 상황을 보여준다.



<그림 7.7> close의 기본 동작: 즉시 리턴

우리는 클라이언트의 data가 도착했을 때, 서버가 일시적으로 바쁜 상태이고, 그래서 data가 TCP의 socket receive buffer에 추가되었다고 가정한다. 비슷하게, 다음 segment인, 클라이언트의 FIN 또한, socket receive buffer에 추가된다 (어떤 방식으로든지 구현은 FIN이 연결을 통해 수신되었다는 것을 기록한다). 그렇지만 기본적으로, 클라이언트의 close는 즉시 리턴한다. 우리가 이 시나리오에서 보여주듯이, 클라이언트의 close는 서버가 자신의 socket receive buffer 내부에서 남은 data를 읽기 전에 리턴할 수 있다. 그러므로, 서버 호스트가 서버 application이 남은 data를 읽기 전에 crash되고, 클라이언트 application이 이것을 절대 알지 못할 가능성이 있다.

클라이언트는 SO_LINGER socket 옵션을 설정해서, 어떤 양수의 linger time을 지정할 수 있다. 이 경우, 클라이언트의 close는 클라이언트의 모든 data와 FIN이 서버 TCP에 의해 확인되기 전까지는 리턴하지 않는다. 우리는 그림 7.8에서 이것을 보여준다.

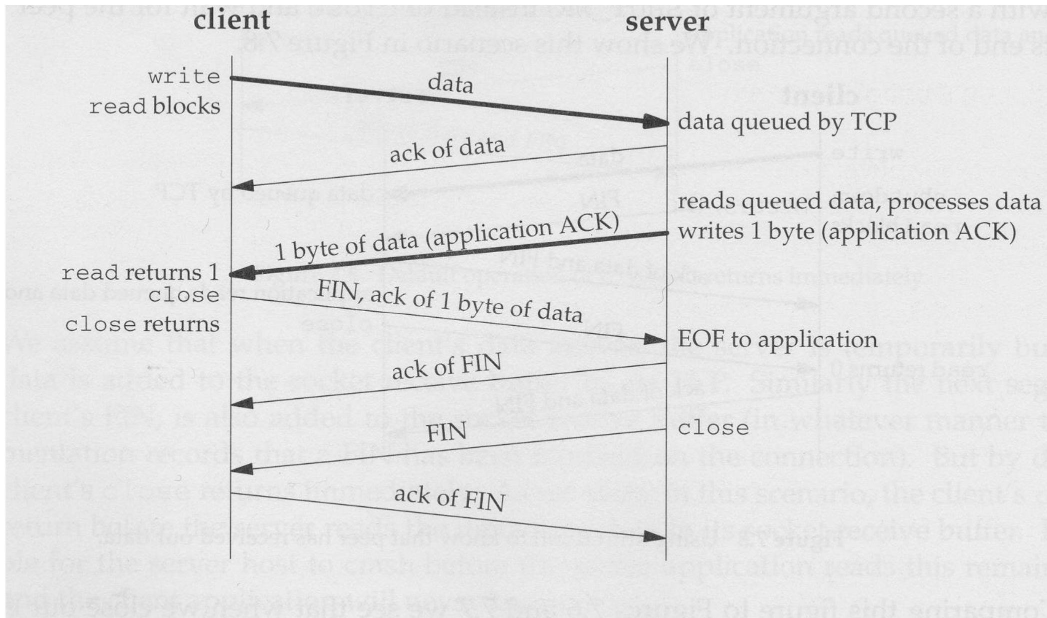


<그림 7.8> SO_LINGER 옵션이 설정되어 있고 l_linger가 양의 값일 때의 close

그렇지만 우리는 여전히 그림 7.7에 있는 것과 동일한 문제를 가지고 있다: 서버 host는 서버 응용이 남은 data를 읽기 전에 crash될 수 있고, 클라이언트 application은 이것을 절대 알지 못할 것이다.

여기에 있는 기본적인 원칙은, SO_LINGER socket 옵션이 설정되어 있을 때, close가 성공적으로 반환되었다는 것은 우리에게 우리가 송신한 data (그리고 FIN)가 peer TCP에 의해 확인되었다는 것 만을 알려준다. peer application이 data를 읽었는지에 대해서는 알려주지 않는다. 만약 우리가 SO_LINGER socket 옵션을 설정하지 않으면, 우리는 peer TCP가 data를 확인했는지 알 수 없다.

서버가 data를 읽었다는 것을 클라이언트가 알 수 있는 방법 중의 하나는 close 대신 shutdown을 호출하고 (두 번째 argument를 SHUT_WR로 하고) peer가 연결의 끝에서 close를 호출하는 것을 기다리는 것이다. 우리는 그림 7.9에서 이 시나리오를 보여준다.



<그림 7.9> peer가 우리의 data를 수신했다는 것을 알기 위해 shutdown 사용

이 그림을 그림 7.7과 7.8과 비교해 볼 때, 우리는 연결에서 우리 쪽의 end를 닫으면, 호출된 함수(close 또는 shutdown)와 SO_LINGER socket 옵션이 설정되었는지에 따라서, 리턴이 세 가지 다른 경우로 발생할 수 있다는 것을 알게 된다:

1. close가 대기 시간 없이, 즉시 리턴한다 (기본 방식, 그림 7.7).
2. close는 우리의 FIN에 대한 ACK을 받을 때까지 남아있다 (그림 7.8).
3. read 다음에 오는 shutdown은 우리가 peer의 FIN을 받을 때까지 기다린다 (그림 7.9).

peer application이 우리의 data를 읽었는지 알 수 있는 다른 방법은 application-level ACK, 또는 application ACK 을 사용하는 것이다. 예를 들어, 다음과 같이, 클라이언트는 data를 서버에게 송신하고 read로 한 byte의 data를 기다린다:

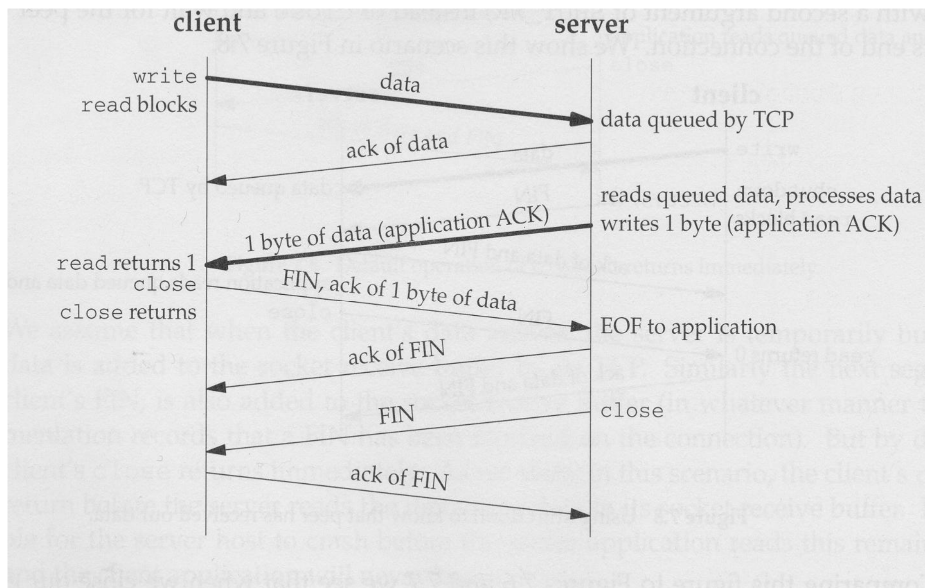
```
char ack;
Write(sockfd, data, nbytes);     /* data from client to server */
n = Read(sockfd, &ack, 1);       /* wait for application-level ACK */
```

서버는 클라이언트로부터 온 data를 읽고 한 byte의 application-level ACK을 전송한다:

```
nbytes = Read(sockfd, buff, sizeof(buff));    /* data from client */
      /* server verifies it received correct amount of data from client */
Write(sockfd, "", 1);           /* server's ACK back to client */
```

우리는 클라이언트의 read가 리턴했을 때, 서버 process가 우리가 보낸 데이터를 받았다는 것을 보장받았다. (서버가 클라이언트가 어느 정도의 data를 송신하는지 알고, 또는 여기에서 우리가 보여주지 않는, 어떤 application-defined end-of-record marker가 있다는 것을 가정한다.) 여기에, application-level ACK은 0 byte지만, 이 byte의 내용은 다른 상태를 서버로부터 클라이언트에게 전달하는 데 사용될 수 있다.

그림 7.10은 가능한 packet exchange를 보여준다.



<그림 7.10> Application ACK

SO_OOBINLINE Socket Option

이 옵션이 설정되면, out-of-band data가 normal input queue(i.e. inline)에 위치하게 될 것이다. 이 경우, 송신 함수의 MSG_OOB flag는 out-of-band 데이터를 받기 위해 사용될 수 없다.

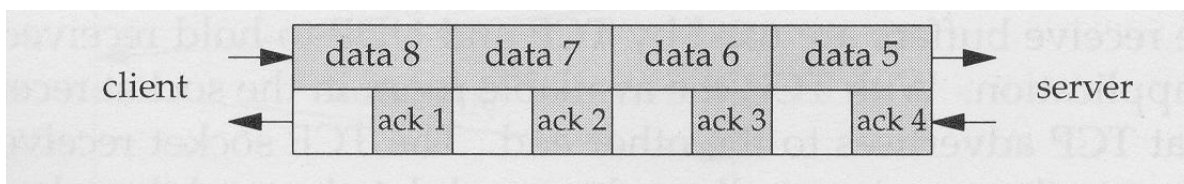
SO_RCBUF and SO_SNDBUF Socket Options

모든 socket은 send buffer와 receive buffer를 가지고 있다. 우리는 2장에서 TCP, UDP, 그리고 SCTP의 send buffer들의 동작 방식에 대해 기술했다.

receive buffer들은 application이 data를 읽기 전까지 수신 받은 데이터를 가지고 있는 데 사용된다. TCP에서, socket receive buffer 내부의 사용 가능한 공간은 TCP가 다른 쪽 end에 알려줄 (advertise) 수 있는 window를 제한한다. TCP socket receive buffer는 peer가 advertised window를 넘는 data를 송신하도록 허가 받지 않았기 때문에 overflow될 수 없다. 이것이 TCP의 flow control이고, 만약 peer가 advertised window를 무시하고 window를 넘어선 data를 송신하면, 수신 받는 TCP는 이 data를 버린다. UDP에선, 그렇지만, socket receive buffer에 맞지 않는 datagram이 도착하면, 그 datagram은 버려진다. UDP는 flow control을 가지지 않는 다는 사실은 기억하라. 빠른 송신자가 느린 수신자를 압도해서, 수신자는 UDP에 의해 datagram들이 버려질 수 있다. 사실, 빠른 송신자는 자신의 network interface를 압도해서, 송신자 자신에 의해 datagram들이 버려지게 할 수 있다.

이 두 socket 옵션들은 우리가 기본 크기들을 변경할 수 있게 해 준다. 기본 값은 구현들에 따라 많은 차이가 있다. 오래된 Berkeley-derived 구현들은 TCP send와 receive buffer들의 기본값을 4,096 byte로 정했을 것이지만, 최근의 system들은 8.192에서 61,440 byte사이의 어떤 값이든, 더 큰 값을 사용한다. UDP send buffer size는 만약 host가 NFS를 지원한다면 기본값을 9,000 byte 근처로 할 것이고, UDP receive buffer size는 보통 기본값이 40,000 byte 근처이다.

TCP socket receive buffer의 크기를 설정할 때, 함수 호출의 순서를 정하는 것이 중요하다.



<그림 7.11> 여덟 개의 segment를 가질 수 있는 TCP 연결(pipe)

SO_RCVTIMEO and SO_SNDTIMEO Socket Options

이 두 socket 옵션들은 socket 수신과 송신에서 timeout을 두도록 해 준다. 두 socketopt 함수들의 argument는, select에 사용되었던 argument와 마찬가지로 timeval structure의 포인터라는 데 주목하라. 이 structure는 우리가 timeout을 second와 microsecond로 지정할 수 있게 해 준다. 우리는 structure의 값을 0 second와 0 microsecond로 설정함으로써 timeout을 비활성화시킬 수 있다. 기본적으로 두 timeout들 모두 비활성화된다.

수신 timeout은 다음의 다섯 input 함수들에 영향을 준다: read, readv, recv, recvfrom, 그리고 recvmsg. 송신 timeout은 다음의 다섯 output 함수들에 영향을 준다: write, writev, send, sendto, sendmsg.

SO_REUSEADDR and SO_REUSEPORT Socket Options

SO_REUSEADDR socket 옵션은 네 가지 다른 용도로 사용된다:

1. SO_REUSEADDR은 전형적으로 다음과 같은 상황에서 사용된다:

   (a) listening server가 시작된다.

   (b) 연결 요청이 도착하고 이 클라이언트를 처리하기 위한 child process가 생성된다.

   (c) listening server가 종료된다. 그렇지만 child는 남아있는 연결에서 계속 클라이언트를 서비스한다.

   (d) listening server가 재시작된다.

   기본적으로, listening server가 socket, bind, 그리고 listen을 호출함으로써 (d)에서 재시작될 때, bind 호출은 listening server가 존재하는 연결(이전에 생성된 child에 의해 관리되는)의 일부인 포트를 bind하려고 시도하기 때문에 실패한다. 그렇지만 만약 server가 socket과 bind 호출 사이에 SO_REUSEADDR socket 옵션을 설정하면, 뒤의 함수는 성공할 것이다. 모든 TCP server들은 이 상황에서 server가 재시작되는 것을 허용하기 위해 이 socket 옵션을 지정해야 한다.

2. SO_REUSEADDR은 각 instance가 서로 다른 local IP address를 bind하는한 새 server가 wildcard address에 bind된 server로써 동일한 포트에서 시작되는 것을 허용해 준다.

   TCP를 사용할 때, 우리는 동일한 IP address와 동일한 port를 bind: completely duplicate binding 하는 여러 개의 서버들을 절대로 시작시킬 수 없다. 이 말은, 우리가 두 번째 서버를

위해 SO_REUSEADDR socket 옵션을 설정했다 하더라도, 198.69.10.2 port 80을 bind하는 서버를 시작하고 또한 198.69.10.2 port 80을 bind하는 다른 서버를 시작할 수 없다는 뜻이다.

보안상의 이유로, 몇몇 operating system들은 이미 wildcard address로 bind 되어 있는 port로 "더 상세한" bind를 하는 것을 제한하고 있다, 이 말은, 여기에서 기술된 순서로의 bind는 SO_REUSEADDR을 설정하든 하지 않든 동작하지 않을 것이라는 의미이다. 그런 system에서, wildcard bind를 수행하는 서버는 반드시 마지막에 시작되어야 한다. 이것은 어떤 악의적인 서버가 어떤 system service에 의해 사용되고 있는 IP address와 port를 bind해서 정당한 요청들을 가로채는 문제를 피하기 위해서이다. 일반적으로 privileged port를 사용하지 않는 NFS에서 특별히 문제가 된다.

3. SO_REUSEADDR은 각 bind가 서로 다른 local IP address를 지정하는 한, 단일 process가 같은 port들 여러 개의 socket들에 bind하는 것을 허용해 준다. 이것은 IP_RECVDSTADDR socket 옵션을 지원하지 않는 system에서 클라이언트 요청들의 destination IP address를 알 필요가 있는 UDP server들에서 일반적이다.

   TCP server들은 연결이 establish된 후에 getsockname을 호출함으로써 destination IP address를 언제나 알 수 있기 때문에 이 기술은 일반적으로 TCP server들에서는 사용되지 않는다. 그렇지만, multi-homed host에 속해 있는 address들에 연결을 제공해 주기를 원하는 TCP server는 이 기술을 사용해야 한다.

4. SO_REUSEADDR은 completely duplicate binding를 허용한다: transport protocol이 이것을 지원할 때, 같은 IP address와 port가 이미 다른 socket에 bind되어 있을 때 이 IP address와 port를 bind하는 것이다. 이 기능은 UDP socket을 위해서만 지원된다.

   이 기능은 같은 host에서 같은 application들이 여러 번 실행되는 것을 허용하기 위해 multicasting과 함께 사용된다. UDP datagram이 이런 여러 번 bind된 socket들 중 하나를 위해 도착하면, 만약 그 datagram이 broadcast address나 multicast address가 목적지라면, 이 datagram의 복사본 하나가 각각의 맞는 socket으로 전달된다. 그렇지만 만약 이 datagram이 unicast address가 목적지라면, 이 datagram에 맞는 socket은 여러 개가 있어서, 어떤 socket이 datagram을 수신해야 하는 지에 대한 결정은 implementation-dependent이다.

우리는 이런 socket 옵션들에 대한 논의를 다음과 같은 권고로 요약할 수 있다:

1) 모든 TCP 서버들에서 bind를 호출하기 전에 SO_REUSEADDR socket 옵션을 설정하라.

2) 같은 host에서 같은 시간에 여러 번 실행될 수 있는 multicast application을 작성할 때, SO_REUSE socket 옵션을 설정하고 그룹의 multicast address를 local IP address로 bind하라.

SO_REUSEADDR에는 잠재적인 보안 문제가 있다. 만약 예를 들어, wildcard address와 port 5555로 bind되어 있는 socket이 존재하고, 만약 우리가 SO_REUSEADDR을 지정하면, 우리는 같은 port를 다른 IP address, 예를 들어 host 의 primary IP address로 bind 할 수 있다. port 5555와 우리가 우리의 socket에 bind한 IP address가 목적지인 모든 미래의 datagram들은 wildcard address에 bind된 다른 socket이 아닌, 우리의 socket으로 전달될 것이다. 이 datagram들은 TCP SYN segment들, SCTP INIT chunk들, 또는 UDP datagram들일 수 있다.

가장 잘 알려진 service들, HTTP, FTP 그리고 Telnet에서, 예를 들어, 이런 서버들은 모두 예약된 port를 bind하기 때문에 이것은 문제가 되지 않는다. 따라서, 뒤따라 오는 process가 그 port의 더 상세한 instance를 bind하려 시도한다면 (i.e., port를 훔치는 것) superuser의 권한이 필요하다. NFS는, 그렇지만, 이것의 normal port (2049)가 예약되어 있지 않았기 때문에 문제가 될 수 있다.

SO_TYPE Socket Option

이 옵션은 socket type을 반환한다. 반환되는 정수값은 SOCK_STREAM 또는 SOCK_DGRAM같은 값이다. 이 옵션은 어떤 process가 시작될 때 socket을 상속받을 경우 사용된다.

7.4 IPv4 Socket Options

이 socket 옵션들은 IPv4에 의해 처리되고 IPPROTO_IP의 level을 가진다.

IP_HDRINCL Socket Option

만약 raw IP socket용으로 이 옵션이 설정되면, 우리는 raw socket으로 송신하는 모든 datagram들에 대해 독자적인 IP header를 만들어야만 한다. 일반적으로, raw socket을 통해 송신되는 모든 datagram들은 kernel이 IP header를 만들어주지만, 몇몇 application들은 IP가 특정 header field들에 설정하는 값들을 덮어쓰기 위해 독자적인 IP header를 만든다.

이 옵션이 설정되면, 우리는 완전한 IP header들 만들 수 있지만, 다음의 예외들이 있다:

- IP는 언제나 IP header checksum을 계산하고 저장한다.
- 만약 우리가 IP identification field를 0으로 설정하면, kernel에서 이 필드를 설정할 것이다.
- IP 옵션들을 설정하는 것은 implementation-dependent하다.
- 몇몇 field들은 host byte order 이고, 몇몇은 network byte order로 있어야 한다.

IP_OPTIONS Socket Option

이 옵션을 설정하는 것은 우리가 IPv4 header 내부에서 IP 옵션들을 설정할 수 있게 해 준다. 이 옵션은 IP header 내부의 IP 옵션들의 형식에 대한 해박한 지식을 필요로 한다.

IP_RECVDSTADDR Socket Option

이 socket 옵션은 수신 받은 UDP datagram의 destination IP address가 recvmsg에 의해 ancillary data로 반환되도록 한다.

IP_RECVIF Socket Option

이 socket 옵션은 UDP datagram을 수신 받은 interface의 index를 recvmsg에 의해 보조 데이터로 반환되도록 한다.

IP_TOS Socket Option

이 옵션은 우리가 TCP, UDP, 또는 SCTP socket을 위해 IP header 내부의 type-of-service field를 설정할 수 있게 해 준다. 만약 우리가 이 옵션을 위해 getsockopt를 호출하면, IP header 내부에서 DSCP와 ECN field들에 위치해 있을 현재 값들 (기본값은 0)이 반환된다. 수신 받은 IP datagram의 값을 가져올 방법은 존재하지 않는다.

application은 prearranged service를 받기 위해 DSCP를 network service provider와 협상한 값으로 설정할 수 있다. RFC 2474에 정의되어 있는 diffserv architecture는, 기존의(historical) TOS field 정의에 대해 제한된 backward compatibility만을 제공한다. IP_TOS 를 <netinet/ip.h>의 내용들 (예를 들어 IPTOS_LOWDELAY 또는 IPTOS_THROUGHPUT)으로 설정하는 application들은, 사용자가 지정한(user-specified) DSCP 값들을 대신 사용해야 한다.

IP_TTL Socket Option

이 옵션을 사용하면, 우리는 system이 주어진 socket으로 송신되는 unicast packet들에 사용하는 기본 TTL을 설정하고 가져올 수 있다. (multicast TTL은 IP_MULTICAST_TTL socket 옵션을 사용한다.) 예를 들어 4.4BSD에서는, TCP와 UDP socket용으로 64의 기본값을 사용하고 raw socket용으로 255를 사용한다. TOS field에서 와 마찬가지로, getsockopt를 호출하면 system이 outgoing datagram들에 사용할 field의 기본값이 반환될 것이다. 수신 받은 datagram에서 값을 얻을 수 있는 방법은 없다.

ICMP6_FILTER Socket Option

이 socket 옵션은 ICMPv6에 의해 처리되고 IPPROTO_ICMPV6의 level을 가진다.

이 옵션은 256개의 가능한 ICMPv6 message type들 중 어떤 것이 raw socket 위의 process로 전달될 것인지를 지정하는 icmp6_filter structure를 가져오고 설정할 수 있게 해 준다.

7.5 IPv6 Socket Options

이 socket 옵션들은 IPv6에 의해 처리되고 IPPROTO_IPV6의 level을 가진다. 모든 IPv6 socket 옵션들은 RFC 3493과 RFC3542에 정의되어 있다.

IPV6_CHECKSUM Socket Option

이 socket 옵션은 checksum field 가 위치해 있는 사용자 데이터의 byte offset을 지정한다. 만약 이 값이 non-negative라면, kernel은: (i) 모든 outgoing packet들에 대해 checksum을 계산하고 저장할 것이다. 그리고, (ii) input 시에 수신받은 checksum을 확인해서, 유효하지 않은 checksum을 가진 패킷들을 버릴(discard) 것이다. 이 옵션은 ICMPv6 raw socket들을 제외하고, 모든 IPv6 socket들에 영향을 미친다. (kernel은 항상 IPv6 raw socket들을 위해 checksum을 계산하고 저장한다.) 만약 -1의 값이 지정되면(기본값), kernel은 이 raw socket을 통해 나가는 outgoing packet들에 대해 checksum을 계산하거나 저장하지 않을 것이고 수신 받은 packet들에 대해 checksum을 검사하지 않을 것이다.

IPv6를 사용하는 모든 protocol들은 자신의 protocol header에 checksum을 가지고 있을 것이다. 이런 checksum들은 source IPv6 address를 checksum의 일부로 포함하는 (일반적으로 raw socket을 IPv4와 사용하게 구현된 모든 다른 protocol들과는 다르게) pseudoheader (RFC 2460)을 포함하고 있다. application이 source address selection을 수행하기 위해 raw socket을 사용하는 것을 강제하기 보다, kernel이 이것을 하고 checksum을 계산하고 저장해서 standard IPv6 pseudoheader에 통합시킬 것이다.

IPV6_DONTFRAG Socket Option

이 옵션을 설정하면 UDP와 raw socket들을 위한 fragment header의 자동 삽입이 비활성화된다. 이 옵션이 설정되었을 때, outgoing interface의 MTU보다 큰 output packet들은 drop될 것이다. packet이 path MTU en-route를 초과할지도 모르기 때문에, packet을 송신하는 system call로부터 어떤 에러도 반환될 필요가 없다. 대신에, application은 path MTU 변화에 대해 알기 위해서 IPV6_RECVPATHMTU 옵션을 활성화시켜야 한다.

IPV6_NEXTHOP Socket Option

이 옵션은 socket address structure로 어떤 datagram을 위한 next-hop address를 지정하고, privileged operation이다.

IPV6_PATHMTU Socket Option

이 옵션은 설정될 수는 없고, 가져올 수만 있다. 이 옵션을 가져오면, path-MTU discovery로 결정된 current MTU가 반환된다.

IPV6_RECVDSTOPTS Socket Option

이 옵션을 설정하면 수신 받은 모든 IPv6 destination option들이 recvmsg에 의해 ancillary data로 반환되도록 지정한다. 이 옵션은 기본값으로 OFF이다.

IPV6_RECVHOPLIMIT Socket Option

이 옵션을 설정하면 수신 받은 hop limit field가 recvmsg에 의해 ancillary data로 반환되도록 지정된다. 이 옵션은 기본값으로 OFF이다.

IPv4가 수신 받은 TTL field를 얻을 수 있는 방법은 없다.

IPV6_RECVHOPOPTS Socket Option

이 옵션을 설정하면 수신 받은 모든 IPv6 hop-by-hop options들이 recvmsg에 의해 ancillary data로 반환되도록 지정된다. 이 옵션은 기본값으로 OFF이다.

IPV6_RECVPATHMTU Socket Option

이 옵션을 설정하면 어떤 path에 대한 path MTU가 변경될 경우 recvmsg에 의해 ancillary data로 반환되도록 지정한다 (어떤 추가 data도 없이).

IPV6_RECVPKTINFO Socket Option

이 옵션을 설정하면 수신 받은 어떤 IPv6 datagram에 대해 다음 두 개의 정보가 recvmsg에 의해 ancillary data로 반환되도록 지정된다: destination IPv6 address와 arriving interface index.

IPV6_RECVRHDR Socket Option

이 옵션을 설정하면 수신 받은 IPv6 routing header가 recvmsg에 의해 ancillary data로 반환되도록 지정된다. 이 옵션은 기본값으로 OFF이다.

IPV6_RECVTCLASS Socket Option

이 옵션을 설정하면 수신 받은 traffic class (DSCP와 ECN field들을 포함하고 있는) 가 recvmsg에 의해 ancillary data로 반환되도록 지정된다. 이 옵션은 기본값으로 OFF이다.

IPV6_UNICAST_HOPS Socket Option

이 IPv6 옵션은 IPv4 IP_TTL socket 옵션과 유사하다. 이 옵션을 설정하면 socket을 통해 송신되는 outgoing datagram들에 대한 default hop limit을 지정하고, 이 socket 옵션을 가져오면 kernel이 socket을 위해 사용할 hop limit의 값이 반환된다. 수신받은 IPv6 datagram에 대한 실제 hop limit field는 IPV6_RECVHOPLIMIT socket 옵션을 사용해서 얻을 수 있다..

IPV6_USE_MIN_MTU Socket Option

이 옵션을 1로 설정하면 path MTU discovery를 수행하지 않고 송신되는 packet들은 fragmentation을 피하기 위해 minimum IPv6 MTU를 사용해서 송신되도록 지정된다. 이 옵션을 0으로 설정하면 모든 destination에 대해 path MTU discovery가 수행된다. 이 옵션을 -1로 설정하면 unicast destination들에 대해 path MTU discovery를 수행하지만 multiple destination들로 송신할 때 minimum MTU를 사용하도록 지정된다. 기본값으로 -1이다.

7.6 TCP SOCKET OPTIONS

TCP 용으로는 두 개의 socket 옵션들이 있다. 우리는 level을 IPPROTO_TCP로 지정한다.

TCP_MAXSEG Socket Option

이 socket 옵션은 우리가 TCP 연결을 위한 MSS를 가져오거나 설정할 수 있게 해 준다. 반환되는 값은 우리의 TCP가 다른 end로 보낼 수 있는 data의 최대량이다; 보통, 이 값은 우리의 TCP가 peer가 통보해 온 MSS 값보다 더 작은 값을 선택하지 않는 한, 다른 end에서 SYN과 함께 통보해 온 MSS값이다. 만약 이 값을 socket이 연결되기 전에 가져오면, 반환되는 값은 만약 다른 end로부터 MSS 옵션이 도착하지 않았을 경우에 사용될 기본값이다.

또한 알아두어야 할 것으로 반환된 값보다 작은 값은 만약, 예를 들어, timestamp 옵션이 사용중일 때 실제로 사용된다, 왜냐하면 이 옵션은 각 segment에서 TCP 옵션의 12 byte를 차지하기 때문이다. 만약 TCP가 path MTU discovery를 사용할 경우 우리의 TCP가 각 segment 당 송신할 수 있는 data의 최대 양은 또한 연결이 지속되는 동안 변할 수 있다. 만약 peer로의 route가 변경될 경우, 이 값은 올라가거나 내려갈 수 있다.

우리는 이 socket 옵션은 또한 application에 의해 설정될 수 있다고 했다. 이것은 모든 system에서 가능한 것은 아니다; 이 옵션은 원래 read-only 옵션이었다. 4.4BSD는 application이 이 값을 감소시키도록 제한한다. 우리는 값을 증가시킬 수 없다. 이 옵션이 TCP가 한 segment당 송신할 수 있는 data의 양을 제어하기 때문에, application이 이 값을 증가시키지 못하게 금지하는 것은 바람직하다. 연결이 establish되면, 이 값은 peer에 의해 통보된 MSS 옵션이고, 우리는 이 값을 초과할 수 없다. 우리의 TCP는, 그렇지만, peer가 통보한 MSS보다 작은 segment는 언제나 송신할 수 있다.

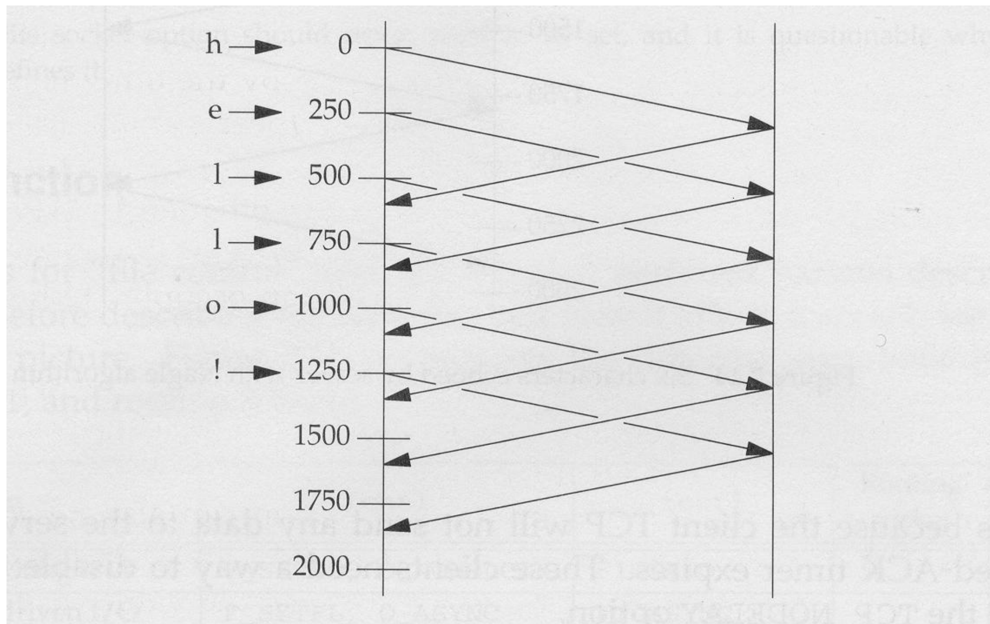
TCP_NODELAY Socket Option

만약 설정되면, 이 옵션은 TCP의 Nagle algorithm을 비활성화시킨다. 기본 설정으로, 이 알고리즘은 활성화 되어있다.

Nagle algorithm의 목적은 WAN에서 small packet들의 수를 줄이는 것이다. algorithm에서는 만약 주어진 연결이 outstanding data를 가지고 있다면 (i.e., 우리의 TCP가 송신했고, 현재 확인을 기다리고 있는 data), 존재하는 data가 확인되기 전까지 연결에서 어떤 small packet들도 사용자의

write 작업에 대한 응답으로 송신될 수 없다고 기술하고 있다. "small" packet의 정의는 MSS보다 작은 모든 packet을 의미한다. TCP는 언제나 가능한 한 최대 크기의 packet을 송신한다; Nagle algorithm의 목적은 연결에서 언제라도 여러 개의 small packet들이 outstanding되는 것을 방지하는 것이다. small packet들을 생성하는 두 가지 예로 Rlogin과 Telnet 클라이언트들이 있다, 이들은 일반적으로 각 keystroke를 분리된 packet으로 송신하기 때문이다. fast LAN에서, 우리는 이런 클라이언트를 사용하면서 Nagle algorithm을 알아차리지 못한다, 왜냐하면 small packet이 확인되기까지 필요한 시간은 보통 몇 millisecond이기 때문이다 - 우리가 두 글자를 연속적으로 타이핑하는 시간보다 훨씬 짧다. 그렇지만 small packet이 확인되기까지 몇 초가 걸리는 WAN에서, 우리는 character echoing에 delay가 있음을 알아차릴 수 있고, 이 delay는 보통 Nagle algorithm에 의해 과장된다.

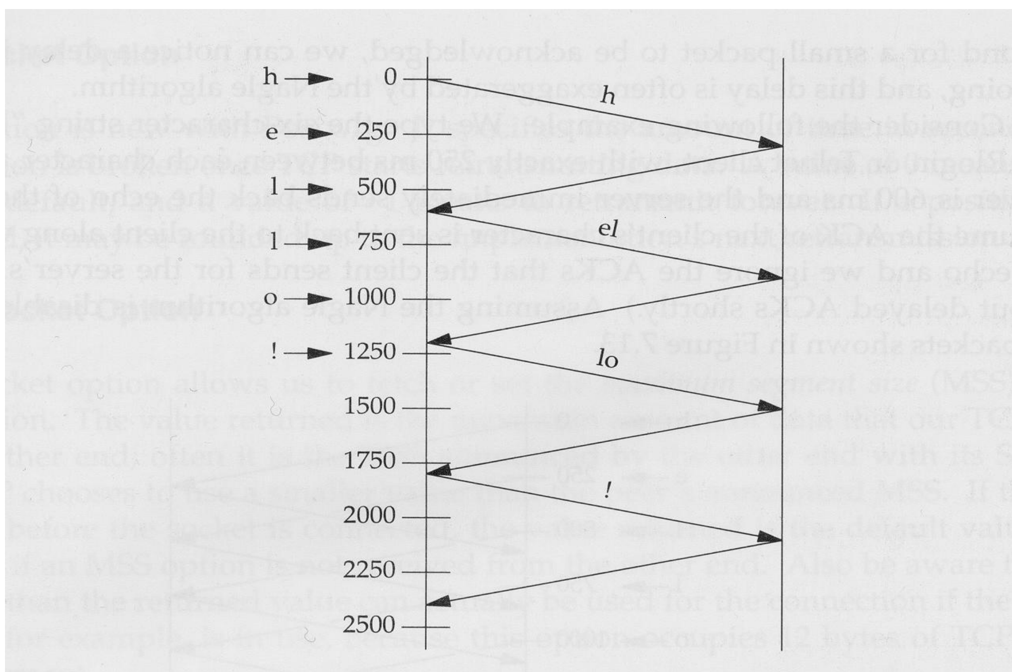
다음의 예를 고려해 보자: 우리는 여섯 글자의 문자열 "hello!"를 Rlogin과 Telnet 클라이언트 모두에, 각 문자 당 정확히 250ms 간격으로 타이핑했다. 서버로의 RTT는 600 ms이고 서버는 각 문자의 echo를 즉시 송신한다. 우리는 클라이언트의 문자에 대한 ACK이 문자 echo와 함께 클라이언트에게 다시 송신되었다고 가정하고 클라이언트가 서버의 echo에 대한 응답으로 송신하는 ACK들은 무시한다. (우리는 잠시 후 delayed ACK들에 대해 이야기할 것이다.) Nagle algorithm이 비활성화되었다고 가정하면, 우리는 그림 7.12에서 보여지는 것처럼 12개의 packet들을 얻는다.



<그림 7.12> Nagle algorithm이 비활성화된 서버로부터 여섯 개의 문자들이 echo된다.

각 문자는 그 자체로 패킷을 통해 송신된다: data segment들은 왼쪽에서 오른쪽으로, 그리고 ACK 들은 오른쪽에서 왼쪽으로 보내진다.

만약 Nagle algorithm이 활성화되면 (기본 동작), 우리는 그림 7.13에 보이는 것처럼 여덟 개의 packet들을 얻는다. 첫 번째 문자는 그 자체로 packet이 되어 송신되지만, 다음의 두 문자들은 송신되지 않는다, 연결은 outstanding중인 small packet을 가지고 있기 때문이다. time 600에, 첫 번째 문자의 echo와 함께, 첫 번째 packet에 대한 ACK이 수신되었을 때, 이 두 packet들이 송신된다. 이 packet이 time 1200에 ACK되기 전까지는, 어떤 small packet들도 더 이상 송신될 수 없다.



<그림 7.13> Nagle algorithm이 활성화된 서버로부터 여섯 개의 문자들이 echo된다.

Nagle algorithm은 보통 다른 TCP algorithm과 상호 작용한다: delayed ACK algorithm. 이 algorithm은 TCP가 data를 수신 받았을 경우 ACK을 바로 송신하지 않도록 한다; 대신에, TCP는 일정 간격의 시간 (보통 50-200 ms) 동안 기다린 후 ACK만을 송신할 것이다. 이 작은 간격의 시간 동안, peer로 다시 전송할 data가 있을 것이고, ACK는 data와 함께 piggyback되어, 한 개의 TCP segment를 절약할 수 있다는 희망이 있다. 이것은 보통 Rlogin과 Telnet 클라이언트의 경우이다, 왜냐하면 서버는 대체로 클라이언트에 의해 송신된 각 문자를 echo하고, 클라이언트의 문자에 대한 ACK은 그 문자에 대한 서버의 ACK과 piggyback되기 때문이다.

문제는 ACK들이 piggyback될 수 있는 반대 방향으로의 traffic을 생성하지 않는 서버를 가진 다른 클라이언트들이다. 이런 클라이언트들은 서버의 delayed ACK timer가 만기될 때까지 클라이언트 TCP가 어떤 data도 송신하지 않을 것이기 때문에 눈에 띄는 delay를 탐지할 수 있다. 이런 클라이언트들은 Nagle algorithm을 비활성화시킬 방법, TCP_NODELAY 옵션이 필요하다.

Nagle algorithm과 TCP의 delayed ACK 과 나쁘게 상호 작용하는 다른 형태의 클라이언트는 single logical request를 작은 조각들로 서버에게 송신하는 클라이언트이다. 예를 들어, 어떤 클라이언트가 400 byte의 request를 자신의 서버로 송신하는 데, 4 byte의 request type 뒤에 396 byte의 request data가 있다고 가정해 보자. 만약 클라이언트가 4-byte write 뒤에 396-byte write를 수행했다면, 두 번째 write는 서버 TCP가 4-byte write를 확인하기 전까지는 클라이언트 TCP에 의해 전송되지 않을 것이다. 또한, 서버 application은 남은 396 byte의 data를 받기 전까지는 4 byte의 data를 가지고 작업을 할 수 없기 때문에, 서버 TCP는 4 byte data의 ACK의 지연시킬 것이다 (i.e., 서버에서 클라이언트로 ACK과 piggyback되는 data는 존재하지 않을 것이다).

이런 형태의 클라이언트를 고치는 방법으로 세 가지 방법이 있다:

1. write를 두 번 호출하는 대신 writev() 를 사용한다. writev를 한 번 호출하는 것은 TCP output을 두 번 호출하는 대신 한번 호출하는 것으로 끝나서, 그 결과 우리의 예에서 하나의 TCP segment가 된다. 이것이 우선적인 해결책이다.

2. 4 byte의 data와 396 byte의 data를 하나의 buffer에 저장한 뒤 이 buffer에 write를 한번 호출한다.

3. TCP_NODELAY socket 옵션을 설정하고 계속 write를 두 번 호출한다. 이것은 최소한의 desirable solution이고, network에 해롭다. 일반적으로 이 방법은 고려되지도 않는다.

7.7 SCTP SOCKET OPTIONS

SCTP를 위한 비교적 많은 수의 socket 옵션들은 (현재 쓰기로 17개) SCTP가 application 개발자에게 제공하는 우수한 상태의 제어를 반영한다. 우리는 level을 IPPROTO_SCTP로 지정한다.

SCTP에 관한 정보를 얻는 데 사용되는 몇몇 옵션들은 data가 kernel로 전달될 것을 요구한다 (e.g., association ID 그리고/또는 peer address). getsockopt의 몇몇 구현들은 data를 kernel내부로 또는 외부로 전달하는 것을 지원하지만, 모두는 아니다. SCTP API에는 이 차이를 숨기기 위해 sctp_opt_info 함수가 정의되어 있다. getsockopt가 이것을 지원하는 system에서, 이 함수는 간단하게 getsockopt의 wrapper이다. 그렇지 않으면, 이 함수는 필요한 동작을 수행한다, 어쩌면 독자적인 ioctl 또는 새로운 system call을 사용한다. 최대한의 portability를 위해 우리는 이런 옵션들을 가져올 때 항상 scto_opt_info를 사용할 것을 추천한다.

SCTP_ADAPTION_LAYER Socket Option

association initialization 중에, 양쪽 endpoint 모두 adaptation layer indication을 지정할 수도 있다. 이 indication은 32-bit unsigned integer이고 어떤 local application adaptation layer라도 통합하기 위해 두 application에 의해 사용될 수 있다. 이 옵션은 caller가 이 endpoint가 peer들에게 제공할 adaptation layer indication을 가져오고 설정하게 해 준다.

이 값을 가져올 때, caller는 local socket이 모든 future peer들에게 제공할 값만을 가져올 수 있을 것이다. peer의 adaptation layer indication을 가져오기 위해, application은 adaptation layer event들을 통보 받아야 한다.

SCTP_ASSOCINFO Socket Option

SCTP_ASSOCINFO socket 옵션은 세 가지 목적으로 사용될 수 있다: (i) 존재하는 association에 대한 정보를 가져오는 데, (ii) 존재하는 association의 parameter를 변경하는 데, 그리고/또는 (iii) 미래의 association을 위한 기본값들을 설정하기 위해. 존재하는 association에 대한 정보를 가져올 때, sctp_opt_info 함수가 getsockopt 대신 사용되어야 한다. 이 옵션은 sctp_assocparams structure를 입력으로 받는다.

```
struct sctp_assocparams {
  sctp_assoc_t sasoc_assoc_id;
  u_int16_t sasoc_asocmaxrxt;
  u_int16_t sasoc_number_peer_destinations;
  u_int32_t sasoc_peer_rwnd;
  u_int32_t sasoc_local_rwnd;
  u_int32_t sasoc_cookie_life;
};
```

sasoc_assoc_id는 관심있는 association의 identification을 가지고 있다. 만약 setsockopt 함수를 호출할 때 이 값이 0으로 설정되어 있으면, sasoc_asocmaxrxt와 sasoc_cookie_life는 socket에서 기본값으로 설정될 값들을 나타낸다. getsockopt를 호출하면 만약 association ID가 제공되면 association-specific 정보들이 반환될 것이다; 그렇지 않고, 만약 이 field가 0이면, 기본 endpoint 설정이 반환될 것이다.

sasoc_asocmaxrxt는 association이 재전송을 포기하고, peer가 unusable하다는 것을 보고하고 association을 닫기 전에 확인 없이 수행할 재전송의 최대 회수를 가지고 있다.

sasoc_number_peer_destinations는 peer destination address들의 수를 가지고 있다. 이 값은 설정될 수 없고, 가져오는 것만 가능하다.

sasoc_peer_rwnd는 현재까지 측정된 peer의 receive window를 가지고 있다. 이 값은 아직까지 송신할 수 있는 data byte들의 총 수를 나타낸다. 이 field는 동적이다; local endpoint가 data를 송신하면, 이 값은 감소한다. remote application이 수신 받는 data를 읽으면, 이 값은 증가한다. 이 값은 이 socket 옵션 호출에 의해 변경될 수 없다.

sasoc_local_rwnd는 SCTP stack이 현재까지 peer에게 보고하고 있는 loca receive window를 나타낸다. 이 값 또한 동적이고 SO_SNDBUF socket 옵션에 영향을 받는다. 이 값은 socket 옵션 호출에 의해 변경될 수 없다.

sasoc_cookie_life는 어떤 remote peer로부터 받는 cookie가 언제까지 유효한지를 millisecond의 수로 나타낸다. peer에게 송신된 각각의 state cookie는 replay attack들을 방지하기 위해 자신과 관계된 lifetime을 가지고 있다. 기본값은 60,000 millisecond이고 이 옵션의 sasoc_assoc_id 값을 0으로 설정함으로써 변경될 수 있다.

SCTP_AUTOCLOSE Socket Option

이 옵션은 우리가 SCTP endpoint를 위해 autoclose time을 가져오거나 설정할 수 있게 해 준다. autoclose time은 SCTP association이 idle일 때 open 상태로 남아있어야 하는 초의 수이다. Idle은 user data를 송신하거나 수신하는 endpoint가 아닌 SCTP stack에 의해 정의된다. autoclose 기능은 기본적으로 비활성화 되어있다.

autoclose 옵션은 one-to-many-style SCTP interface에서 사용되도록 계획되었다 (9장). 이 옵션이 설정되면, 이 옵션으로 전해지는 정수는 idle connection이 close되기 전의 초의 수이다; 0의 값은 autoclose를 비활성화시킨다. 이 endpoint에서 생성된 future association들만이 이 옵션에 영향을 받을 것이다; 기존의 association들은 자신들의 현재 설정을 유지한다.

Autoclose는 서버가 추가적인 상태를 관리할 필요 없이 idle association들을 close하도록 강제하기 위해 서버에 의해 사용될 수 있다. 이 기능을 사용하는 서버는 자신의 모든 association들에서 예상되는 가장 긴 idle time을 주의 깊게 평가할 필요가 있다. autoclose 값을 필요한 값보다 작게 설정하면 association들의 premature closing이 발생할 것이다.

SCTP_DEFAULT_SEND_PARAM Socket Option

SCTP는 보통 ancillary data로 전달되거나 또는 sctp_sendmsg 함수 호출 (보통 사용자를 위해 ancillary data를 전달하는 라이브러리 호출로 구현되는)과 함께 사용될 수 있는 많은 부가적인 send parameter들을 가지고 있다. 많은 수의 message들을, 동일한 parameter들과 함께 송신하고자 하는 application은, 기본 parameter들을 설정하고 ancillary data 또는 sctp_sendmsg 호출을 사용하는 것을 피하기 위해 이 옵션을 사용할 수 있다. 이 옵션은 sctp_sndrcvinfo structure를 입력으로 받는다.

```
struct sctp_sndrcvinfo {
  u_int16_t sinfo_stream;
  u_int16_t sinfo_ssn;
  u_int16_t sinfo_flags;
  u_int32_t sinfo_ppid;
  u_int32_t sinfo_context;
  u_int32_t sinfo_timetolive;
  u_int32_t sinfo_tsn;
  u_int32_t sinfo_cumtsn;
  sctp_assoc_t sinfo_assoc_id;
};
```

sinfo_stream은 모든 message들이 송신될 새 기본 stream을 지정한다. sinfo_ssn의 기본 옵션들을 설정할 때는 무시된다. recvmsg 함수 또는 sctp_recvmsg 함수를 사용해서 message를 수신받을 때, 이 field는 peer가 SCTP DATA chunk 내부의 stream sequence number (SSN) field에 넣은 값을 가지고 있을 것이다.

sinfo_flags는 모든 미래의 message 송신들에 적용될 기본 flag들을 알려준다. sinfo_pid는 모든 data 전송에서 SCTP payload protocol identifier를 설정할 때 사용할 기본값을 제공한다.

sinfo_context는 peer로 송신할 수 없는 message들을 가져올 때 local tag로써 제공될, sinfo_context field에 놓을 기본 값을 지정한다.

sinfo_timetolive는 모든 message 송신들에 적용될 기본 lifetime을 알려준다. lifetime field는 과다한 delay로 인해 (첫 번째 전송 이전에) outgoing message를 언제 버릴 것인지 알기 위해 SCTP stack들에서 사용된다. 만약 두 endpoint들이 partial reliability 옵션을 지원한다면, lifetime은 또한 어떤 message가 첫 번째 전송 이후 언제까지 유효한지를 지정하기 위해 사용된다.

sinfo_tsn은 기본 옵션들을 설정할 때는 무시된다. recvmsg 함수 또는 sctp_recvmsg 함수로 message를 수신할 때, 이 field는 peer가 SCTP DATA chunk 내부의 transport sequence number (TSN) field에 넣은 값을 가지고 있을 것이다.

sinfo_cumtsn은 기본 옵션을 설정할 때는 무시된다. recvmsg 함수 또는 sctp_recvmsg 함수로 message를 수신할 때, 이 field는 local SCTP stack이 자신의 remote peer와 결합한 현재까지의 cumulative TSN을 가지고 있을 것이다.

sinfo_assoc_id는 요청자가 기본 parameter에 대응해 설정되기를 원하는 association identification을 지정한다. one-to-one socket들에서 이 field는 무시된다.

모든 기본 설정들은 자체적인 sctp_sndrcvinfo structure를 가지지 않는 message들에만 영향을 미친다는 것을 적어둔다. 이 structure를 제공하는 송신들 (e.g., ancillary data를 가지 sctp_sendmsg 또는 sendmsg 함수) 은 기본 설정을 덮어쓸 것이다. 기본값들을 설정하는 것과 별개로, 이 옵션은 sctp_opt_info 함수를 사용해서 현재까지의 기본 parameter들을 가져오는 데 사용될 수도 있다.

SCTP_DISABLE_FRAGMENTS Socket Option

SCTP는 일반적으로 하나의 SCTP packet에 들어가지 않는 모든 user message들을 fragment시켜서 여러 개의 DATA chunk들로 넣는다. 이 옵션을 설정하면 송신자의 이런 동작을 비활성화시킨다. 이 옵션에 의해 비활성화되면, SCTP는 EMSGSIZE를 반환하고 message를 송신하지 않을 것이다. 이 옵션의 기본 동작은 비활성화되는 것이다.

이 옵션은 모든 사용자 application message가 하나의 IP packet에 맞도록 하는 것으로, message 크기를 제어하기를 원하는 application들에 의해 사용될 수도 있다.

SCTP_EVENTS Socket Option

이 socket 옵션은 호출자가 다양한 SCTP notification들을 가져오고, 활성화시키고, 또는 비활성화시킬 수 있게 해 준다. SCTP notification은 SCTP stack이 application으로 송신할 message이다. 이 message는 recvmsg 함수의 msg_flags field가 MSG_NOTIFICATION으로 설정되면, 일반적인 data로 읽혀진다. recvmsg와 sctp_recvmsg를 사용할 준비가 되어 있지 않은 application은 event들을 활성화시키지 말아야 한다.

이 옵션을 사용하고 sctp_event_subscribe structure를 전달하는 것으로 여덟 개의 다른 형태의 event들이 전달될(subscribed) 수 있다. 0의 값은 non-subscription을 나타내고 1의 값은 subscription을 나타낸다.

sctp_event_subscribe structure는 다음의 형태를 가진다:

```
struct sctp_event_subscribe {
    u_int8_t sctp_data_io_event;
    u_int8_t sctp_association_event;
    u_int8_t sctp_address_event;
    u_int8_t sctp_send_failure_event;
    u_int8_t sctp_peer_error_event;
    u_int8_t sctp_shutdown_event;
    u_int8_t sctp_partial_delivery_event;
    u_int8_t sctp_adaptation_layer_event;
};
```

SCTP_GET_PEER_ADDR_INFO Socket Option

이 옵션은 congestion window, smoothed RTT와 MTU를 포함한, peer address에 관한 정보를 가져온다. 이 옵션은 특정 peer address에 관한 정보를 가져오는 데에만 사용될 수 있을 것이다. 호출자는 sctp_paddrinfo structure를 spinfo_address field를 관심있는 peer address로 채워서 제공하고, maximum portability를 위해 getsockopt 대신 sctp_opt_info를 사용해야 한다. sctp_paddrinfo structure는 다음의 형식을 가진다:

```
struct sctp_paddrinfo {
  sctp_assoc_t spinfo_assoc_id;
  struct sockaddr_storage spinfo_address;
  int32_t spinfo_state;
  u_int32_t spinfo_cwnd;
  u_int32_t spinfo_srtt;
  u_int32_t spinfo_rto;
  u_int32_t spinfo_mtu;
};
```

호출자에게 반환되는 data는 다음을 제공한다:

- spinfo_assoc_id는 association identification 정보를 포함하고 있고, 이것은 또한 "communication up" nitofication (SCTP_COMM_UP)에서도 제공된다. 이 유일한(unique) 값은 거의 모든 SCTP 동작들을 위해 association을 표현하는 빠른 방법으로 사용될 수 있다.
- spinfo_address는 SCTP socket에게 어떤 address로 정보를 반환해야 하는지 알려주기 위해 호출자에 의해 설정된다. 반환될 때, 이 값은 변경되면 안 된다.
- spinfo_cwnd는 peer address를 위해 기록된 현재의 congestion window를 나타낸다.
- spinfo_srtt는 이 address를 위해 현재까지 측정된 smoothed RTT를 나타낸다.
- spinfo_rt는 이 address를 위해 사용중인 현재의 retransmission timeout을 나타낸다.
- spinfo_mtu는 path MTU discovery 에 의해 발견된 현재까지의 path MTU를 나타낸다.

이 옵션을 사용하는 흥미로운 방법 중 하나는 IP address structure를 다른 호출들에 사용될 수 있는 association identification으로 바꾸는 것이다.

SCTP_I_WANT_MAPPED_V4_ADDR Socket Option

이 flag는 AF_INET6-type socket 위의 IPv4-mapped address들을 활성화시키거나 비활성화시키는데 사용될 수 있다. 활성화되면 (기본 동작방식), 모든 IPv4 address들은 application으로 전달되기 전에 IPv6 address로 매핑될 것이다. 만약 이 옵션이 비활성화되면, SCTP socket은 IPv4 address들을 매핑하지 않을 것이고 대신 그 address들을 sockaddr_in structure로 전달할 것이다.

SCTP_INITMSG Socket Option

이 옵션은 INIT message를 송신할 때 SCTP socket에서 사용되는 기본 initital parameter들을 가져오거나 설정하는 데 사용될 수 있다. 옵션은 sctp_initmsg structure를 사용하고, 이 structure는 다음과 같이 정의된다:

```
struct sctp_initmsg {
  uint16_t sinit_num_ostreams;
  uint16_t sinit_max_instreams;
  uint16_t sinit_max_attempts;
  uint16_t sinit_max_init_timeo;
};
```

이 field들은 다음과 같이 정의된다:

- sinit_num_ostreams는 application이 요청하길 원하는 outbound SCTP stream들의 수를 나타낸다. 이 값은 association이 initial handshake를 끝내기 전까지는 승인되지 않고, peer endpint limitation들을 통해 아래쪽으로(downward) 협상될 수 있다.
- sinit_max_instreams는 application이 허용할 준비가 된 inbound stream들의 최대 수를 나타낸다. 만약 이 값이 SCTP stack이 지원하는 최대 허용 가능한 stream보다 크다면 SCTP stack에 의해 덮어씌어질 수 있다.
- sinit_max_attempts는 peer endpoint가 unreachable하다는 것으로 간주하기 전에 SCTP stack이 initial INIT message를 몇 번이나 보내야 하는지를 표현한다.

- sinit_max_init_timeo는 INIT timer를 위한 최대 RTO 값을 나타낸다. initial timer의 exponential backoff동안, 이 값은 재전송의 최고 한도로 RTO.max를 대체한다. 이 값은 millisecond로 표현된다.

SCTP_MAXBURST Socket Option

이 socket 옵션은 application이 packet들을 송신할 때 사용되는 maximum burst size를 가져오거나 설정할 수 있게 해 준다. SCTP 구현이 data를 peer에게 송신할 때, packet으로 network를 flooding시키는 것을 피하기 위해 SCTP_MAXBURST packet들만이 즉시 송신된다. 구현은 이 한계를 다음과 같이 사용한다: (i) congestion window 를 current flight size 더한 후 maximum burst size에 path MTU 를 곱한 것으로 줄인다, 또는 (ii) 이 값을 separate micro-control로 사용해서, 적어도 maximum burst packet들을 송신할 기회가 생길 때마다 전송한다.

SCTP_MAXSEG Socket Option

이 socket 옵션은 application이 SCTP fragmentation이 이루어지는 동안 maximum fragment size를 가져오고 설정할 수 있게 해 준다. 이 옵션은 TCP 옵션 TCP_MAXSEG 와 유사하다.

SCTP 송신자가 application으로부터 이 값보다 큰 message를 받으면, SCTP 송신자는 peer endpoint로의 전송을 위해 message를 여러 개의 조각으로 나눌 것이다. SCTP 송신자가 일반적으로 사용하는 크기는 peer와 연관된 모든 address에서 가장 작은 MTU이다. 이 옵션은 이 값을 지정된 값을 향해 아래쪽으로(downward) 덮어쓴다. SCTP stack은 이 옵션에 의해 요청된 것 보다 더 작은 boundary로 message를 fragment 할 수도 있다는 것을 적어둔다. 이런 더 작은 fragmentation은 peer endpoint로의 path들 중 하나가 SCTP_MAXSEG 옵션에서 요청된 값보다 작은 MTU를 가지고 있을 경우 발생한다.

SCTP_NODELAY Socket Option

만약 이 옵션이 설정되면, SCTP의 Nagle algorithm을 비활성화시킨다. 이 옵션은 기본적으로 OFF이다 (i.e., Nagle algorithm은 기본적으로 ON이다). SCTP의 Nagle algorithm은 단순하게 stream에 byte들을 합치는 것과는 다르게 여러 개의 DATA chunk를 합치려 한다는 것을 제외하고 TCP의 것과 동일하게 동작한다. Nagle algorithm에 대한 더 자세한 설명은 TCP_MAXSEG를 보라.

SCTP_PEER_ADDR_PARAMS Socket Option

이 socket 옵션은 application이 association에 관한 다양한 parameter들을 가져오거나 설정할 수 있게 해 준다. 호출자는 sctp_paddrparams structure에 association identification의 채워서 제공한다.

sctp_paddrparams structure는 다음의 형식을 가진다:

```
struct sctp_paddrparams {
  sctp_assoc_t spp_assoc_id;
  struct sockaddr_storage spp_address;
  u_int32_t spp_hbinterval;
  u_int16_t spp_pathmaxrxt;
};
```

이 field들은 다음과 같이 정의된다:

- spp_asoc_id는 요청되거나 설정될 정보를 위한 association identification의 가지고 있다. 만약 이 값이 0으로 설정되면, endpoint 기본값이 association-specific 값들 대신 설정되거나 가져와진다.
- spp_address는 이 parameter들이 요청되거나 설정되는 IP address를 지정한다. 만약 spp_assoc_id field가 0으로 설정되어 있으면, 이 field는 무시된다.
- spp_hbinterval의 heartbeat들 사이의 간격이다. SCTP_NO_HB의 값은 heartbeat들을 비활성화시킨다. SCTP_ISSUE_HB의 값은 on-demand heartbeat를 요청한다. 모든 다른 값들은 millisecond 단위로 heartbeat interval을 이 값으로 바꾼다. 기본 parameter들을 설정할 때, SCTP_ISSUE_HB은 값은 허용되지 않는다.
- spp_hbpathmaxrxt는 이 목적지가 INACTIVE로 선언되기 전에 시도될 재전송의 회수를 가지고 있다. 어떤 address가 INACTIVE로 선언되고, 이 address가 primary address이면, 대체 address가 primary로써 선택될 것이다.

SCTP_PRIMARY_ADDR Socket Option

이 socket 옵션은 local endpoint가 primary로 사용하는 address를 가져오거나 설정한다. primary address는, 기본적으로, peer에게 보내는 모든 message의 destination address로 사용된다. 이 값을 설정하기 위해, 호출자는 association identification과 primary address로 사용될 peer의 address를 채워 넣어야 한다. 호출자는 이 정보를 sctp_setprim structure를 통해 전달하고, 그 structure는 다음과 같이 정의된다:

```
struct sctp_setprim {
        sctp_assoc_t          ssp_assoc_id;
        struct sockaddr_storage    ssp_addr;
};
```

이 field들은 다음과 같이 정의된다:

- ssp_assoc_id는 요청자가 설정하거나 가져오기를 원하는 현재 primary address의 association identification을 지정한다. one-to-one 형식에서, 이 field는 무시된다.
- sspp_addr은 반드시 peer에게 속해있는 address 여야 하는, primary address를 지정한다. 만약 동작이 setsockopt 함수 호출이면, 이 field의 값은 요청자가 primary destination address로 만들고자 하는 새 peer address를 나타낸다.

하나의 연관된 local address만을 가진 one-to-one socket에서 이 옵션의 값을 가져오는 것은 getsockname을 호출하는 것과 동일하다는 것을 적어둔다.

SCTP_RTOINFO Socket Option

이 socket 옵션은 이 endpoint에서 사용되는 특정 association 또는 기본 값들에서 다양한 RTO 정보들을 가져오거나 설정하는 데 사용될 수 있다. 정보를 가져올 때, 호출자는 maximum portability를 위해 getsockopt 대신 sctp_opt_info를 사용해야 한다. 호출자는 다음의 형식으로 sctp_rtoinfo structure를 제공한다:

```
struct sctp_rtoinfo {
        sctp_assoc_t  srto_assoc_id;
        uint32_t      srto_initial;
        uint32_t      srto_max;
        uint32_t      srto_min;
};
```

- srto_assoc_id는 관심 있는 특정 association 또는 0을 가지고 있다. 만약 이 field가 0의 값을 포함하고 있으면, system의 기본 파라메터는 이 호출에 영향을 받는다.

- stro_initial의 peer address에 사용되는 initial RTO 값을 포함하고 있다. initital RTO는 INIT chunk를 peer에게 송신할 때 사용된다. 이값은 millisecond단위이고 3,000의 기본값을 가지고 있다.

- srto_max는 retransmission timer에 업데이트가 이루어질 때 사용되는 maximum RTO 값을 포함하고 있다. 만약 업데이트된 값이 RTO maximum보다 크면, RTO maximum이 측정된 값 대신 RTO로 사용된다. 이 field의 기본값은 60,000 millisecond이다.

- srto_min은 retransmission timer를 시작할 때 사용될 minimum RTO를 포함하고 있다. RTO timer에 업데이트가 이루어질 때마다, RTO minimum 값은 새 값과 비교해서 확인된다. 만약 새 값이 minimum보다 작으면, minimum이 새 값을 대체한다. 이 field의 기본값은 1,000 millisecond이다.

srto_initial, srto_max, 또는 srto_min에 0의 값이 들어가는 것은 현재 설정된 기본값이 변경되어선 안 된다는 것을 의미한다. 모든 time 값들은 millisecond로 표시된다.

SCTP_SET_PEER_PRIMARY_ADDR Socket Option

이 옵션을 설정하면 peer가 local address의 자신의 primary address로 설정할 것을 요청하는 message가 보내지도록 한다. 호출자는 sctp_setpeerprim structure를 제공해야 하고 peer가 자신의 primary로 표시하도록 요청하기 위해 association adentification과 local address를 채워 넣어야 한다. 제공되는 address는 반드시 local endpoint의 bound address중 하나여야 한다. sctp_setpeerprim structure는 다음과 같이 정의된다:

```
struct sctp_setpeerprim {
       sctp_assoc_t         sspp_assoc_id;
       struct sockaddr_storage    sspp_addr;
};
```

이 field들은 다음과 같이 정의된다:

- sspp_assoc_id는 요청자가 primary address로 설정하기를 바라는 association identification을 지정한다. one-to-one 형식에서, 이 field는 무시된다.
- sspp_addr은 요청자가 peer system이 primary address로 설정하기를 바라는 local address를 가지고 있다.

이 기능은 부가적인 것이고, 동작하기 위해서 반드시 양쪽 endpoint에서 지원되어야 한다. 만약 local endpoint가 이 기능을 지원하지 않으면, EOPNOTSUPP 이 호출자에게 반환될 것이다. 만약 remote endpoint가 이 기능을 지원하지 않으면, EINVAL이 호출자에게 반환될 것이다. 이 값은 설정할 수만 있고 가져올 수는 없다는 것을 적어둔다.

SCTP_STATUS Socket Option

이 socket 옵션은 SCTP association의 현재 상태를 가져올 것이다. 호출자는 maximum portability를 위해 getaddrinfo 대신 sctp_opt_info를 사용해야 할 것이다. 호출자는 sctp_status structure를 제공하고, association identification field, sstat_assoc_id를 채운다. structure는 요청한 association에 적합한 정보가 들어가 있는 상태로 반환될 것이다. sctp_status structure는 다음의 형식을 가지고 있다:

```
struct sctp_status {
        sctp_assoc_t sstat_assoc_id;
        int32_t sstat_state;
        u_int32_t sstat_rwnd;
        u_int16_t sstat_unackdata;
        u_int16_t sstat_penddata;
        u_int16_t sstat_instrms;
        u_int16_t sstat_instrms;
        u_int16_t sstat_instrms;
        struct sctp_paddrinfo sstat_primary;
};
```

이 field들은 다음과 같이 정의된다:

- sstat_assoc_id는 association identification을 가지고 있다.
- sstat_state association에 대한 전반적인 state를 가리킨다.
- sstat_rwnd는 우리의 endpoint의 현재 측정한 peer의 receive window를 가지고 있다.
- sstat_unackdata는 peer를 위해 남겨져 있는 확인되지 않은 DATA chunk들의 수를 가지고 있다.
- sstat_penddata는 local endpoint가 application이 읽게 하기 위해 가지고 있는 읽혀지지 않은 DATA chunk들의 수를 가지고 있다
- sstat_instrms는 peer가 data를 이 endpoint로 송신하기 위해 사용하는 stream들의 수를 가지고 있다.
- sstat_outstrms는 이 endpoint가 data를 peer에게 송신하기 위해 사용할 수 있는 stream들의 수를 가지고 있다.
- sstat_fragmentation_point는 local SCTP endpoint가 사용자 message들을 위한 fragmentation point로 사용하고 있는 현재 값을 포함하고 있다. 이 값은 모든 목적지의 MTU 중 최소값이거나, SCTP_MAXSEG와 함께 local application 에 의해 설정된 더 작은 값일 가능성이 있다.
- sstat_primary는 현재의 primary address를 가지고 있다. primary address는 data를 peer endpoint로 송신할 때 사용되는 기본 address이다.

이 값들은 진단(diagnostics)과 세션의 특성들을 판단하는 데 유용하다; 예를 들어, sctp_get_no_strms 함수는 outbound use를 위해 얼마나 많은 stream들이 사용 가능한지를 판단하기 위해 sstat_outstrms 멤버를 사용할 것이다.

낮은 sstat_rwnd 그리고/또는 높은 sstat_unackdata 값은 peer의 receive socket buffer 가 가득 찼는지 판단하는 데 사용될 수 있고, application이 가능하다면 전송을 늦출 수 있는 신호로 사용된다. sstat_fragmentation_point는 더 작은 application message들을 송신함으로써, 몇몇 application들이 SCTP가 생성해야 하는 fragment들의 수를 줄이게 하는 데 사용된다.

7.8 FCNTL 함수

fcntl은 "file control"을 의미하고 이 함수는 다양한 descriptor 제어 동작들을 수행한다. 함수를 기술하고 이 함수가 socket에 어떤 영향을 주는지 설명하기 전에, 우리는 더 큰 그림을 볼 필요가 있다. 다음 표에 fcntl, ioctl, 그리고 routing socket들에 의해 수행되는 서로 다른 동작들이 요약되어 있다.

Operation	fcntl	ioctl	Routing socket	Posix.1g
set socket for nonblocking I/O	F_SETFL, O_NONBLOCK	FIONBIO		fcntl
set socket for signal-driven I/O	F_SETFL, O_ASYNC	FIOASYNC		fcntl
set socket owner	F_SETOWN	SIOCSPGRP or FIOSETOWN		fcntl
get socket owner	F_GETOWN	SIOCGPGRP or FIOGETOWN		fcntl
get #bytes in socket receive buffer		FIONREAD		
test for socket at out-of-band mark		SIOCATMARK		sockatmark
obtain interface list		SIOCGIFCONF	sysctl	
interface operations		SIOC[GS]IF*xxx*		
ARP cache operations		SIOC*x*ARP	RTM_*xxx*	
routing table operations		SIOC*xxx*RT	RTM_*xxx*	

처음 여섯 개의 동작들은 어떤 process에 의해서든 socket에 적용될 수 있다 다음 두(interface operations) 동작들은 덜 일반적인 것이지만, 여전히 일반적인 용도에 사용된다; 그리고 마지막 두 (ARP와 routing table) 동작들은 ifconfig와 route같은 관리용 프로그램들에 의해 발행된다.

처음 네 개의 동작들을 수행하는 데는 다양한 방법들이 있지만, 우리는 첫 번째 열에서 fcntl을 사용하는 것이 우선적인 방법이라고 POSIX에서 기술되어 있다는 것을 적어놓는다. 마지막 열이 공백인 이외의 동작들은, POSIX에 의해 표준화되지 않은 동작들이다.

fcntl 함수는 네트워크 프로그래밍과 관련해서 다음의 기능들을 제공한다:

- Nonblocking I/O – 우리는 socket을 nonblocking으로 설정하기 위해 F_SETFL 명령을 사용해서 O_NONBLOCK filestatus를 설정할 수 있다. 우리는 16장에서 Nonblocking I/O에 대해 기술할 것이다.
- Signal-driven I/O – 우리는 socket의 status가 변경될 때 SIGIO signal이 생성되도록, F_SETFL 명령을 사용해서 O_ASYNC file status를 설정할 수 있다.

- F_SETOWN 명령은 우리가 SIGIO와 SIGURG signal들을 받기 위해 socket owner (process ID 또는 process group ID)를 설정할 수 있게 해 준다.

```
#include <fcntl.h>
int fcntl(int fd, int cmd, … /* int arg */ );
```

OK 일 경우 cmd에 의존한 값을, 에러시 -1을 리턴한다

각 descriptor는 (socket 을 포함해서) F_GETFL 명령으로 가져올 수 있고 F_SETFL 명령으로 설정할 수 있는 file flag들의 집합을 가지고 있다. socket에 영향을 주는 두 flag들은

```
O_NONBLOCK - nonblocking I/O
O_ASYNC - signal-driven I/O
```

우리는 이후에 이 기능들에 대해 자세히 기술할 것이다. 지금은, fcntl을 사용해서, nonblocking I/O를 활성화시키는 전형적인 코드를 다음과 같이 적는다:

```
int     flags;
        /* Set a socket as nonblocking */
if ( (flags = fcntl(fd, F_GETFL, 0)) < 0)
        err_sys("F_GETFL error");
flags |= O_NONBLOCK;
if (fcntl(fd, F_SETFL, flags) < 0)
        err_sys("F_SETFL error");
```

우리는 단지 원하는 flag를 설정만 하는 코드를 접할 수 있다.

```
        /* Wrong way to set a sockt as nonblocking */
if (fcntl(fd, F_SETFL, O_NONBLOCK) < 0)
        err_sys("F_SETFL error");
```

이 코드는 nonblocking flag를 설정하면서, 또한 모든 다른 file status flag들을 지운다. file status flag들 중 하나를 설정하는 올바른 방법은 현재 flag들을 가져와서, 새 flag에 OR 논리연산을 하고, flag를 설정하는 방법뿐이다.

다음의 코드는 nonblocking flag를 끈다. flags는 위에서 보인 fcntl의 호출로 설정되었다고 가정한다.

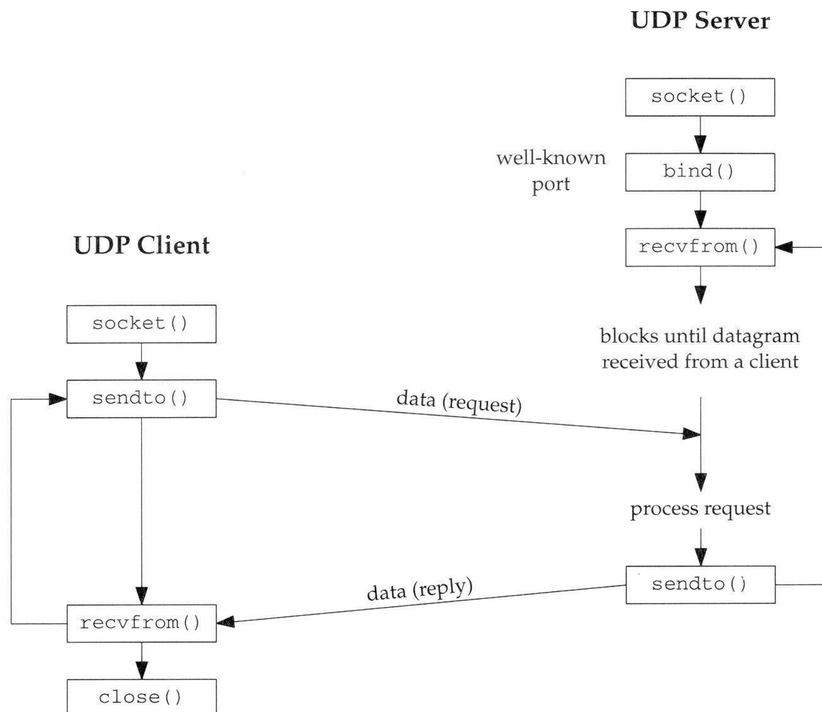
```
flags &= ~O_NONBLOCK;
if (fcntl(fd, F_SETFL, flags) < 0)
        err_sys("F_SETFL error");
```

두 개의 signal SIGIO와 SIGURG는 F_SETOWN명령으로 owner에게 할당된 socket을 위해서만 생성된다는 점에서 다른 signal들과 차이가 있다. F_SETOWN 명령의 위한 정수 arg 값은 signal을 받기 위해 process ID를 지정하는 양의 정수, 또는 signal을 받기 위해 절대값을 process group ID로 지정하는 음의 정수가 될 수 있다. F_GETOWN 명령은 fcntl 함수의 반환값으로써 socket owner를 반환하고, 이 값은 process ID (양의 반환값) 또는 process group ID (-1보다 작은 음수의 값) 이다. signal을 받기 위해 process ID를 지정하는 것과 process group ID를 지정하는 것의 차이는 앞의 것은 단지 하나의 process만이 signal을 받게 하고, 뒤의 것은 process group 내부의 모든 process들 (아마도 하나 이상의)이 signal을 받게 한다.

8. UDP SOCKETS

TCP를 사용해서 작성된 application들과 UDP를 사용해서 작성된 application들 사이에는 몇 가지 기반적인 차이점이 있다. 이 차이점은 두 transport layer들의 차이에서 비롯된다: TCP에서 제공되는 connection-oriented, reliable byte stream과 다르게, UDP는 connectionless, unreliable, datagram protocol이다. 그럼에도 불구하고, TCP 대신 UDP를 사용하는 것이 이치에 맞는 경우들이 있다: 예를 들어 DNS, NFS, 그리고 SNMP 등이다.

그림 8.1은 전형적인 UDP 클라이언트/서버를 위한 함수 호출들을 보여준다. 클라이언트는 서버와 연결을 개설(establish)하지 않는다. 대신, 클라이언트는 sendto를 사용해서 서버에 datagram을 보내고 (다음 절에서 설명됨), 이것은 목적지(서버)의 address를 parameter로 요구한다. 비슷한 것으로, 서버는 클라이언트로부터의 연결을 허용하지 않는다. 대신에, 서버는 단지 recvfrom 함수를 호출하고, 이것은 data가 어떤 클라이언트로부터 올 때까지 기다린다. recvfrom은 클라이언트의 protocol address를 datagram과 함께 반환해서, 서버는 적합한 클라이언트에 응답을 보낼 수 있다.



<그림 8.1> UDP 클라이언트/서버를 위한 socket 함수들

그림은 UDP 클라이언트/서버 교환에서 발생하는 전형적인 시나리오의 timeline을 보여준다. 우리는 이것을 전형적인 TCP 교환과 비교할 수 있다. 이 장에서, 우리는 UDP socket들과 함께 사용할 새 함수들, recvfrom과 sendto에 대해 기술할 것이고, UDP를 사용해서 우리의 echo 클라이언트/서버를 다시 해볼 것이다. 우리는 또한 UDP socket에서 connect 함수의 사용과 asynchronous error의 개념에 대해 기술할 것이다.

8.1 RECVFROM 및 SENDTO 함수

이 두 함수들은 read, write 함수들과 비슷하지만, 세 개의 추가적인 argument들이 필요하다.

```
#include <sys/socket.h>

ssize_t recvfrom(int sockfd, void *buff, size_t nbytes, int flags, struct
sockaddr *from, socklen_t *addrlen);

ssize_t sendto(int sockfd, const void *buff, size_t nbytes, int flags,
const struct sockaddr *to, socklen_t addrlen);
```

OK시 읽거나 쓴 byte의 수, 에러시 -1을 리턴한다

처음 세 argument들, sockfd, buff, 그리고 nbytes는, read와 write에서의 처음 세 argument들과 동일하다: 읽거나 쓸 buffer로의 pointer, 그리고 읽거나 쓸 byte의 수이다. 이 장에서 우리는 flags를 언제나 0으로 설정할 것이다.

sendto의 to argument는 data가 송신될 protocol address (e.g., IP address와 port 번호)를 포함하고 있는 socket address structure이다. 이 socket address structure의 크기는 addrlen에 의해 지정된다. recvfrom 함수는 from에 의해 지정된 socket address structure를 datagram을 송신한 쪽의 protocol address로 채운다. 이 socket address structure에 저장된 byte의 수는 addrlen에 의해 정수 형태로 호출자에게 반환된다. sendto의 마지막 argument는 정수값이지만, recvfrom의 마지막 argument는 정수 값으로의 pointer (value-result argument)라는 것을 적어둔다.

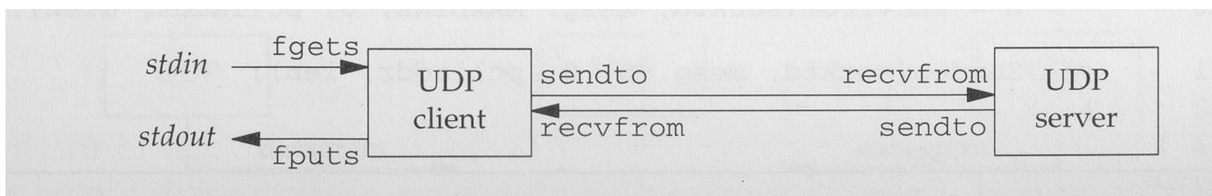
recvfrom의 마지막 두 argument는 accept의 마지막 두 argument들과 비슷하다: 반환되는 socket address structure의 내용은 우리에게 누가 datagram을 송신했는지 (이 경우 UDP) 또는 누가 연결을 시작했는지 (이 경우 TCP) 를 알려줄 것이다. sendto의 마지막 두 argument들은 connect의

마지막 두 argument들과 비슷하다: 우리는 socket address structure를 datagram을 송신할 곳의 protocol address (이 경우 UDP) 또는 누구와 연결을 establish할 것인지 (이 경우 TCP) 로 채운다.

두 함수 모두 이 함수의 값으로 읽혀지거나 쓰여진 data의 길이를 반환한다. recvfrom의 전형적인 사용에서, datagram protocol에서 반환값은 수신 받은 datagram 내부의 사용자 data의 양이다.

8.2 UDP ECHO 서버

우리는 이제 5장에서의 echo 클라이언트/서버를 UDP를 사용해서 다시 해 볼 것이다. 우리의 UDP 클라이언트와 서버 프로그램들은 우리가 그림 8.1에서 그렸던 함수 호출 흐름을 따라갈 것이다. 그림 8.2는 사용되는 함수들을 묘사하고 있다. 그림 8.3은 서버의 main 함수를 보여준다.



<그림 8.2> UDP를 사용한 간단한 echo 클라이언트/서버

udpcliserv/udpserv01.c

```
 1 #include    "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     sockfd;
 6     struct sockaddr_in servaddr, cliaddr;

 7     sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

 8     bzero(&servaddr, sizeof(servaddr));
 9     servaddr.sin_family = AF_INET;
10     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
11     servaddr.sin_port = htons(SERV_PORT);

12     Bind(sockfd, (SA *) &servaddr, sizeof(servaddr));

13     dg_echo(sockfd, (SA *) &cliaddr, sizeof(cliaddr));
14 }
```

udpcliserv/udpserv01.c

<그림 8.3> UDP echo 서버

UDP socket을 생성하고, 서버의 well-known port를 bind하기

우리는 socket의 두 번째 argument를 SOCK_DGRAM (IPv4 protocol 내부의 datagram socket)으로 지정해서 UDP socket을 생성한다. TCP 서버의 예제와 같이, bind를 위한 IPv4 address는 INADDR_ANY로 지정되고 서버의 well-known port는 unp.h 헤더에 있는 SERV_PORT 상수이다.

dg_echo 함수는 서버 동작을 수행하기 위해 호출된다.

UDP Echo 서버: dg_echo 함수

그림 8.4는 dg_echo 함수를 보여준다.

lib/dg_echo.c

```
 1 #include    "unp.h"

 2 void
 3 dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
 4 {
 5     int     n;
 6     socklen_t len;
 7     char    mesg[MAXLINE];

 8     for ( ; ; ) {
 9         len = clilen;
10         n = Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);

11         Sendto(sockfd, mesg, n, 0, pcliaddr, len);
12     }
13 }
```

lib/dg_echo.c

<그림 8.4> dg_echo 함수: datagram socket 위의 echo line

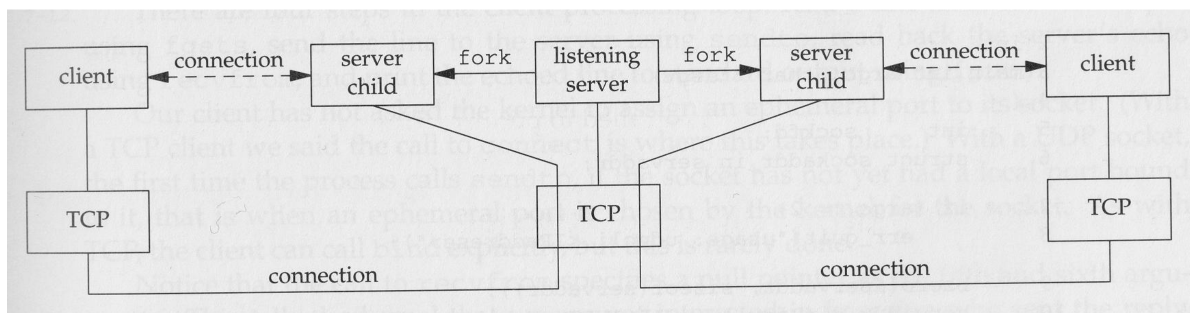
datagram을 읽고, 송신자에게 echo해주기

이 함수는 recvfrom을 사용해서 서버의 port에 도착하는 다음 datagram을 읽어서 sendto를 사용해서 다시 송신해주는 간단한 루프이다. 이 함수가 간단하기는 하지만, 여기에는 고려해야 할 몇 가지 자세한 사항들이 있다. 첫 번째로, 이 함수는 종료되지 않는다. UDP가 connectionless protocol이기 때문에, 우리가 TCP에서 가지고 있던 EOF 같은 것은 존재하지 않는다.

다음으로, 이 함수는 우리가 TCP에서 가지고 있던 concurrnt server가 아닌, interative server를 제공한다. fork의 호출이 없어서, 하나의 서버 process가 모든 클라이언트를 관리한다. 일반적으로, 대부분의 TCP 서버들은 concurrent이고 대부분의 UDP 서버들은 iterative이다.

이 socket을 위해 UDP layer에서는 잠재적인 queuing이 발생하고 있다. 각각의 UDP socket은 receive buffer를 가지고 있고 이 socket으로 도착하는 각각의 datagram은 그 socket receive buffer에 위치한다. process가 recvfrom을 호출하면, buffer에 있는 다음 datagram이 first-in, first-out (FIFO) 순서로 process에게 반환된다. 이 방식으로, 만약 process가 socket을 위해 이미 queue 되어 있는 datagram을 읽을 수 있게 되기 전에 여러 개의 datagram이 socket에 도착하면, 도착하는 datagram들은 단지 socket receive buffer에 추가된다. 그렇지만, 이 buffer는 제한된 크기를 가지고 있다.

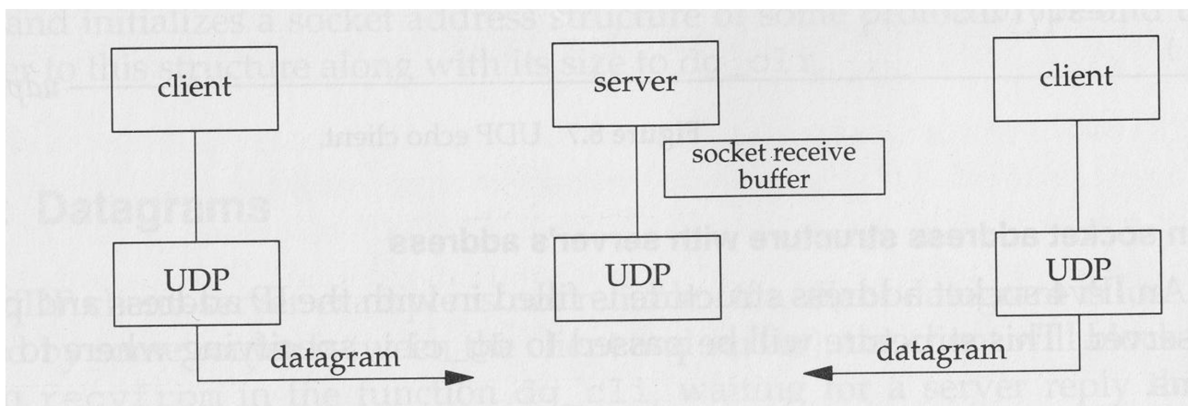
다음 그림은 TCP 클라이언트/서버에서 두 클라이언트가 서버와 연결을 establish하는 경우이다.



<그림 8.5> 두 개의 클라이언트를 가진 TCP 클라이언트/서버의 요약

여기에는 두 개의 연결된 socket들과 자체적인 socket receive buffer를 가지고 있는 서버 host에서의 두 개의 연결된 socket들이 있다.

그림 8.6은 두 클라이언트들이 UDP 서버로 datagram들을 송신할 때의 시나리오를 보여준다.



<그림 8.6> 두 개의 클라이언트를 가진 UDP 클라이언트/서버의 요약.

서버 process는 하나뿐이고 이 process는 도착하는 모든 datagram들을 수신하고 모든 응답들을 송신하는 하나의 socket을 가지고 있다. 이 socket은 도착하는 모든 datagram이 위치하는 receive buffer를 가지고 있다.

그림 8.3의 main 함수는 protocol-dependent 이지만, dg_function은 protocol-independent이다. dg_echo가 protocol-independent인 이유는 호출자 (우리의 경우 main 함수) 는 반드시 socket address structure를 정확한 크기로 할당해야만 하고, 이 structure의 pointer는, structure의 크기와 함께, dg_echo의 argument로 전달된다. 함수 dg_echo는 이 protocol-dependent structure의 내부를 절대로 들여다보지 않는다: 이 함수는 단지 이 structure의 pointer를 recvfrom과 sendto에 전달할 뿐이다. recvfrom은 이 structure를 클라이언트의 IP address와 port 번호로 채운다, 그리고 동일한 pointer(pcliaddr)가 목적지 address로 sendto에게 전달되기 때문에, 이것이 datagram을 송신한 클라이언트로 datagram이 echo되는 방법이다.

8.3 UDP ECHO CLIENT

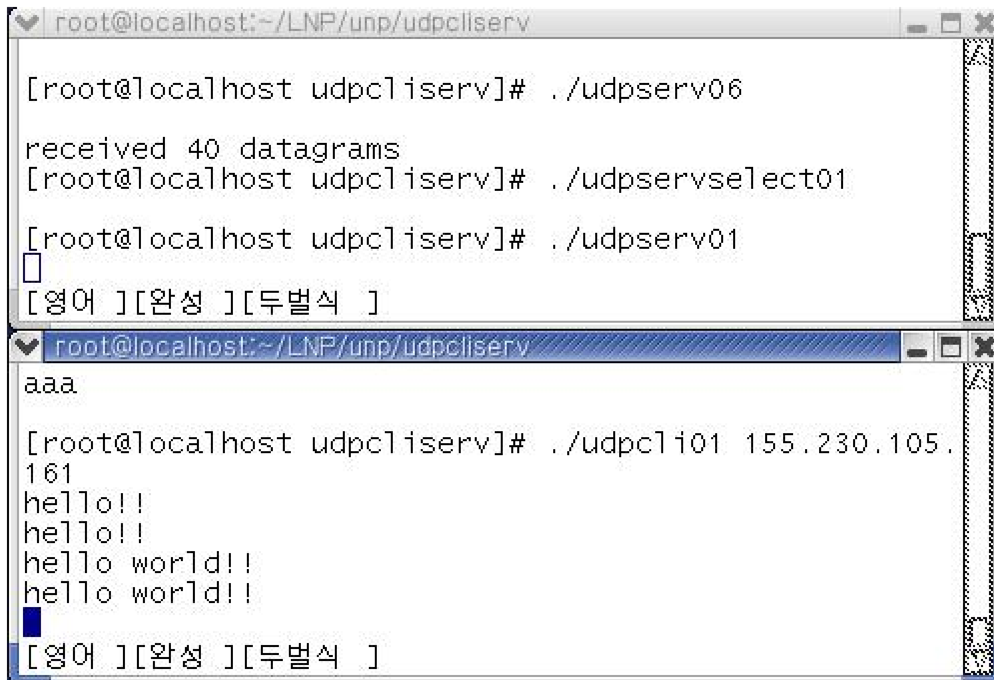
UDP 클라이언트의 main 함수는 그림 8.7에서 볼 수 있다.

udpcliserv/udpcli01.c

```
 1 #include    "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     sockfd;
 6     struct sockaddr_in servaddr;

 7     if (argc != 2)
 8         err_quit("usage: udpcli <IPaddress>");

 9     bzero(&servaddr, sizeof(servaddr));
10     servaddr.sin_family = AF_INET;
11     servaddr.sin_port = htons(SERV_PORT);
12     Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

13     sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

14     dg_cli(stdin, sockfd, (SA *) &servaddr, sizeof(servaddr));

15     exit(0);
16 }
```

udpcliserv/udpcli01.c

<그림 8.7> UDP echo 클라이언트



UDP Echo Client: dg_cli 함수

그림 8.8은 대부분의 클라이언트 작업을 수행하는 dg_cli함수를 보여준다.

lib/dg_cli.c

```
 1 #include    "unp.h"

 2 void
 3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 4 {
 5     int     n;
 6     char    sendline[MAXLINE], recvline[MAXLINE + 1];

 7     while (Fgets(sendline, MAXLINE, fp) != NULL) {

 8         Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

 9         n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);

10         recvline[n] = 0;    /* null terminate */
11         Fputs(recvline, stdout);
12     }
13 }
```

lib/dg_cli.c

<그림 8.8> dg_cli 함수: 클라이언트 processing loop

클라이언트 동작 루프에는 네 단계가 있다: fgets를 사용해서 standard input으로부터 line을 읽어들이고, sendto를 사용해서 line을 서버로 송신하고, recvfrom을 사용해서 서버의 echo를 다시 읽고, fputs를 사용해서 echo된 line을 standard output으로 출력한다.

우리의 클라이언트는 단말(ephemeral) port를 자신의 socket에 할당할 것을 kernel에게 요청하지 않았다. (TCP 클라이언트에서, 우리는 connect의 호출이 이 요청이 발생하는 곳이라고 이야기했다.) UDP socket에서, process가 sendto를 호출하는 첫 번째 시간에, 만약 socket이 bind된 local port를 아직 가지지 않았다면, 그 때가 kernel에 의해 socket을 위한 단말 port가 선택되는 때이다. TCP를 사용할 때처럼, 클라이언트는 명시적으로 bind를 호출할 수 있지만, 이것은 거의 수행되지 않는다.

recvfrom의 호출은 다섯 번째 그리고 여섯 번째 argument로 null pointer를 지정한다는 것을 주목하라. 이것은 kernel에게 우리는 누가 응답을 송신했는지 아는 데에는 관심이 없다는 것을 알려준다. 어떤 process이던지 간에, 같은 host 또는 다른 host 모두에서, datagram을 클라이언트의 IP address와 port로 송신해서, 이 datagram을 읽는 클라이언트가, 이것을 서버의 응답으로 생각할 위험이 있다.

서버 함수 dg_echo에서와 같이, 클라이언트 함수 dg_cli는 protocol-independent하지만, 클라이언트 main함수는 protocol-dependent하다. main 함수는 socket address structure를 어떤 protocol type으로 할당하고 초기화한 뒤 이 structure의 pointer를, structure의 크기와 함께, dg_cli에게 전달한다.

8.4 LOST DATAGRAM

우리의 UDP 클라이언트/서버는 안정적이지 못하다. 만약 클라이언트 datagram이 사라지면 (예를 들어 클라이언트와 서버 사이의 어떤 라우터에서 사라졌다고 하면), 클라이언트는 함수 dg_cli 내부의 recvfrom 호출에서 영원히 block되어, 도착할 수 없는 서버의 응답을 기다리게 될 것이다. 비슷하게, 만약 클라이언트 datagram이 서버에 도착했지만 서버의 응답이 사라지면 클라이언트는 다시 recvfrom 호출에서 영원히 block될 것이다. 이것을 막는 전형적인 방법은 클라이언트의 recvfrom 호출에 timeout을 거는것이다.

단지 recvfrom에 timeout을 거는 것이 완전한 해결책은 아니다. 예를 들어, 만약 우리의 시간이 다 지나면, 우리는 datagram이 서버로 가지 못했는지, 또는 만약 서버의 응답이 오지 못했는지

말할 수 없다. 만약 클라이언트의 요청이 "어느 정도의 돈을 계좌 A에서 계좌 B로 전송하라" 같은 것이었다면 (우리의 간단한 echo 서버 대신), 요청이 사라진 것인지 응답이 사라진 것인지 사이에는 큰 차이가 있을 것이다.

수신된 응답의 검증

우리는 클라이언트의 단말 port 번호를 아는 어떤 process라도 우리의 클라이언트에게 datagram들을 보낼 수 있고, 이 datagram들은 서버의 일반 응답과 섞일 수 있다. 우리가 할 수 있는 일은 우리에게 응답을 보낸 측의 IP address와 port를 반환하고 우리가 datagram을 송신한 서버가 아닌 다른 곳에서 온 모든 datagram을 무시하도록 recvfrom 호출을 변경하는 것이다. 그렇지만, 우리가 보게 될 것으로, 이것에 관해서 몇몇 함정(pitfall)들이 있다.

첫 번째로, 우리는 클라이언트 main 함수를 변경해서 standard echo 서버를 사용하도록 한다. 우리는 대입문을 다음과 같이 변경한다.

```
servaddr.sin_port = htons(SERV_PORT);
servaddr.sin_port = htons(7);
```

우리는 이렇게 함으로써 standard echo 서버가 돌아가는 어떤 host라도 우리의 클라이언트와 함께 사용할 수 있다.

우리는 그 후 dg_cli 함수를 다시 코딩해서 recvfrom에 의해 반환되는 structure를 가지고 있기 위한 다른 socket address structure를 할당한다.

다른 socket address structure 할당하기

우리는 malloc을 호출함으로써 다른 socket address structure를 할당한다. dg_cli 함수는 여전히 protocol-independent하다는 것을 알려준다; 우리는 어떤 형태의 socket structure를 다루고 있는지 신경쓰지 않기 때문에, 우리는 단지 malloc 호출 내부에서 structure의 크기만 사용한다.

반환된 address 비교하기

recvfrom 호출 내부에서, 우리는 kernel에게 datagram 송신자의 address를 반환할 것을 알려준다. 우리는 첫 번째로 recvfrom에서 value-result argument 형식으로 반환된 길이를 비교한 뒤 memcmp를 사용해서 socket address structure 자체를 비교한다.

만약 socket address structure가 length field를 포함하고 있더라도, 우리는 그것을 설정할 필요도 조사할 필요도 없다고 말하고 있다. 그렇지만, memcmp는 두 socket address structure 내부의 모든 data의 byte를 비교하고, length field는 kernel이 반환하는 socket address structure 내부에 설정되어 있다; 그래서 이 경우 sockaddr을 만들 때 우리는 반드시 이 field를 설정해 놓아야만 한다. 만약 우리가 그렇게 하지 않으면, memcmp는 0을 16 (sockaddr_in을 가정해서) 과 비교하고 일치하지 않게 될 것이다.

udpcliserv/dgcliaddr.c

```
 1 #include    "unp.h"

 2 void
 3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 4 {
 5     int     n;
 6     char    sendline[MAXLINE], recvline[MAXLINE + 1];
 7     socklen_t len;
 8     struct sockaddr *preply_addr;

 9     preply_addr = Malloc(servlen);

10     while (Fgets(sendline, MAXLINE, fp) != NULL) {

11         Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

12         len = servlen;
13         n = Recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
14         if (len != servlen || memcmp(pservaddr, preply_addr, len) != 0) {
15             printf("reply from %s (ignored)\n",
16                    Sock_ntop(preply_addr, len));
17             continue;
18         }
19         recvline[n] = 0;        /* null terminate */
20         Fputs(recvline, stdout);
21     }
22 }
```

udpcliserv/dgcliaddr.c

<그림 8.9> 반환된 socket address를 확인하는 버전의 dg_cli.

새 버전의 우리의 클라이언트는 만약 서버가 단일 IP address만을 가진 host 위에 있을 때에는 잘 동작할 것이다. 그렇지만 이 프로그램은 만약 서버가 multihomed 이면 실패할 수 있다. 우리는 이 프로그램을 두 개의 interface와 두 개의 IP address를 가진 우리의 host freebsd4에서 실행한다.

```
macosx % host freebsd4
freebsd4.unpbook.com has address 172.24.37.94
freebsd4.unpbook.com has address 135.197.17.100
macosx % udpcli02 135.197.17.100
hello
reply from 172.24.37.94:7 (ignored)
goodbye
reply from 172.24.37.94:7 (ignored)
```

우리는 같은 subnet을 공유하지 않는 IP address를 클라이언트로 지정한다.

recvfrom에 의해 반환된 IP address (UDP datagram의 출발지 IP address) 는 우리가 datagram을 송신한 측의 IP address가 아니다. 서버가 응답을 송신할 때, 목적지 IP address는 172.24.37.78이다. freebsd4의 kernel 내부에 있는 routing 기능은 172.24.37.93를 outgoing interface로 고른다. 서버가 이 socket을 위해 IP address를 bind하지 않은 관계로 (서버는 자신의 socket에 wildcard address를 bind했고, 이것은 우리가 freebsd에서 netstat을 돌려서 확인할 수 있다), kernel은 IP datagram을 위한 출발지 address를 고른다. 이 address는 outgoing interface의 primary IP address로 선택되었다 (TCPv2의 pp.232-233). 또한 이 address가 interface의 primary IP address인 관계로, 만약 우리가 datagram을 interface의 nonprimary IP address (i.e., an alias)로 송신하면, 이것 또한 Figure 8.9에 있는 우리의 테스트가 실패하게 되는 원인이 될 것이다.

한 가지 해결책은 recvfrom에서 반환된 IP address에 대해, 클라이언트가 DNS 내부의 서버 이름을 봄으로써 응답하는 host의 domain name을 host의 IPaddress 대신 확인하는 것이다. 다른 해결책은 UDP서버가 host에서 설정된 모든 IP address를 위해서 하나의 socket을 생성하고, 그 IP address를 socket에 bind하고, 모든 socket들에 select를 사용하고 (socket들 중 하나가 readable 상태가 되기를 기다림), 그 후 readable 상태인 socket에 응답을 보내는 것이다. 응답에 사용된

socket이 클라이언트의 요청의 목적지 address인 IP address에 bind된 관계로 (그렇지 않으면 datagram은 socket에 도착하지도 못했을 것이다), 이것은 응답의 출발지 address가 요청의 목적지 address와 동일하다는 것을 보장한다.

8.5 SERVER NOT RUNNING

살펴볼 다음 시나리오는 서버를 시작하지 않고 클라이언트를 시작하는 것이다. 만약 우리가 이렇게 하고 한 line을 클라이언트에 입력하면, 아무 일도 일어나지 않는다. 클라이언트는 자신의 recvfrom 호출에서 영원히 block되어, 영원히 오지 않을 서버의 응답을 기다릴 것이다. 그렇지만, 이것은 우리의 networking application에 어떤 일이 일어나고 있는지 이해하기 위해 underlying protocol에 대해 더 많이 이해할 필요가 있는 것의 예시일 뿐이다.

첫 번째로 우리는 host maxosx에서 tcpdump를 시작하고, 그 뒤 같은 host에서 클라이언트를 시작하고, 서버 host로 freebsd4를 지정한다. 우리는 그 후 한 line을 입력한다, 그렇지만 이 line은 echo되지 않는다.

maxosx % udpcli01 **172.24.37.94**

hello, world 우리는 이 line을 입력하지만

아무것도 echo되어 돌아오지 않는다

다음은 tcpdump 출력을 보여준다.

```
0.0          arp who-has freebsd4 tell macosx
0.003576     arp reply freebsd4 is-at 0:40:5:42:d6:de
0.003601     macosx.51139 > freebsd4.9877: udp 13
0.009781     freebsd4 > macosx: icmp: freebsd4 udp port 9877 unreachable
```

첫 번째로 우리는 클라이언트 host가 서버 host로 UDP datagram을 송신하기 전에 ARP request와 reply가 필요하다는 것을 알게 되었다. (우리는 IP datagram이 다른 host 또는 local network의 다른 router로 송신되기 전에 ARP request-reply의 가능성을 반복하기 위해 출력 내부에 이 교환을 남겨둔다.)

3번째 줄에서 우리는 클라이언트 datagram이 송신되었지만 서버 host는 4번째 줄에서 ICMP "port unreachable"로 응답한 것을 볼 수 있다. (길이 13은 12 개의 문자들과 newline을 의미한다.) 이 ICMP 에러는, 그렇지만, 클라이언트 process로 반환되지 않고, 그 이유는 우리가 간략하게 기술할 것이다. 대신에, 클라이언트는 recvfrom 호출 내부에서 영원히 block된다. 우리는 또한 ICMPv6는 ICMPv4와 유사한 "port unreachable" 에러를 가지고 있어서, 여기에서 기술되는 결과는 IPv6에서도 비슷하다.

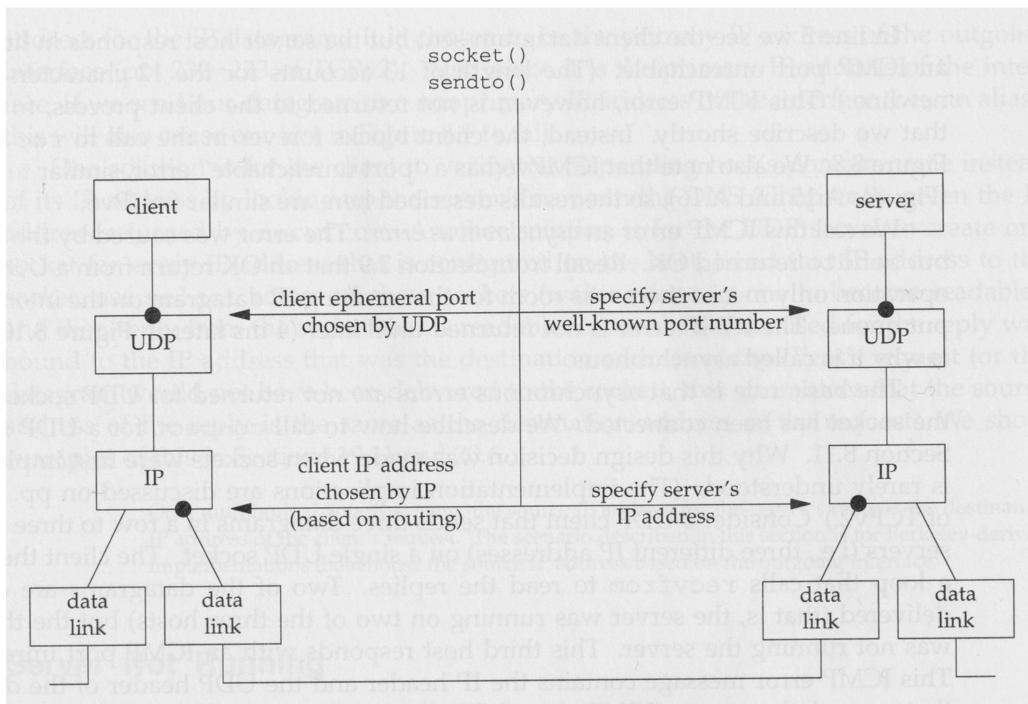
우리는 이 ICMP 에러를 asynchronous error라 부른다. 에러는 sendto 에 의해 발생되지만, sendto는 성공적으로 반환된다. 2절에서 UDP 출력 동작의 성공적인 반환은 interface output queue에 IP datagram을 위한 공간이 있다는 의미일 뿐이라고 했던 것을 기억하라. ICMP 에러는 나중(Figure 8.10에서는 4 ms 뒤)까지 반환되지 않고, 이것이 우리가 이 에러를 asynchronous라고 부르는 이유이다.

기본적인 법칙은 socket연결되어 있지 않다면, asynchronous error는 UDP socket으로 반환되지 않는다는 것이다. 우리는 뒤에서 UDP socket을 위해 connect를 어떻게 호출하는지를 기술할 것이다. socket이 처음 구현될 때 이렇게 설계되도록 결정된 이유는 잘 이해되지 않는다.

하나의 UDP socket 위에서 세 개의 datagram들을 차례대로 세 개의 다른 서버들 (i.e. 세 개의 서로 다른 IP address를 가진)로 송신하는 어떤 UDP 클라이언트를 생각해보자. 클라이언트는 그 후 응답을 읽기 위해 recvfrom을 호출하는 loop로 들어간다. datagram들 중 두 개는 성공적으로 전달되었지만 (이 말은, 세 개의 host들 중 두 곳에서 서버가 돌아가고 있었다) 세 번째 host에서는 서버가 돌아가지 않았다. 이 세 번째 host는 ICMP port unreachable 로 응답한다. 이 ICMP error message는 에러를 발생시킨 datagram의 IP header와 UDP header를 포함하고 있다. (ICMPv4와 ICMPv6 error message들은 ICMP error를 수신하는 측이 어떤 socket이 에러를 발생시켰는지를 판단하게 하기 위해 언제나 IP header와 모든 UDP header 또는 TCP header의 일부를 포함한다) 세 개의 datagram들을 송신한 클라이언트는 이 중 어떤 datagram이 에러를 발생시켰는지 구분하기 위해 에러를 발생시킨 datagram의 목적지를 알 필요가 있다. 그렇지만 kernel이 어떻게 이 정보를 process에게 전달할 수 있는가? recvfrom이 반환할 수 있는 유일한 정보는 errno 값이다; recvfrom은 에러가 발생한 datagram의 목적지 IP address와 목적지 UDP port 번호를 반환할 방법이 없다. 결론은, 따라서, process가 UDP socket을 정확히 하나의 peer에만 연결했을 때만 이 asynchronous error들을 process에게 반환하는 것이다.

8.6 UDP 예제 요약

다음은 클라이언트가 UDP datagram을 송신할 때 반드시 지정하거나 선택해야 하는 네 개의 값들을 보여주고 있다.

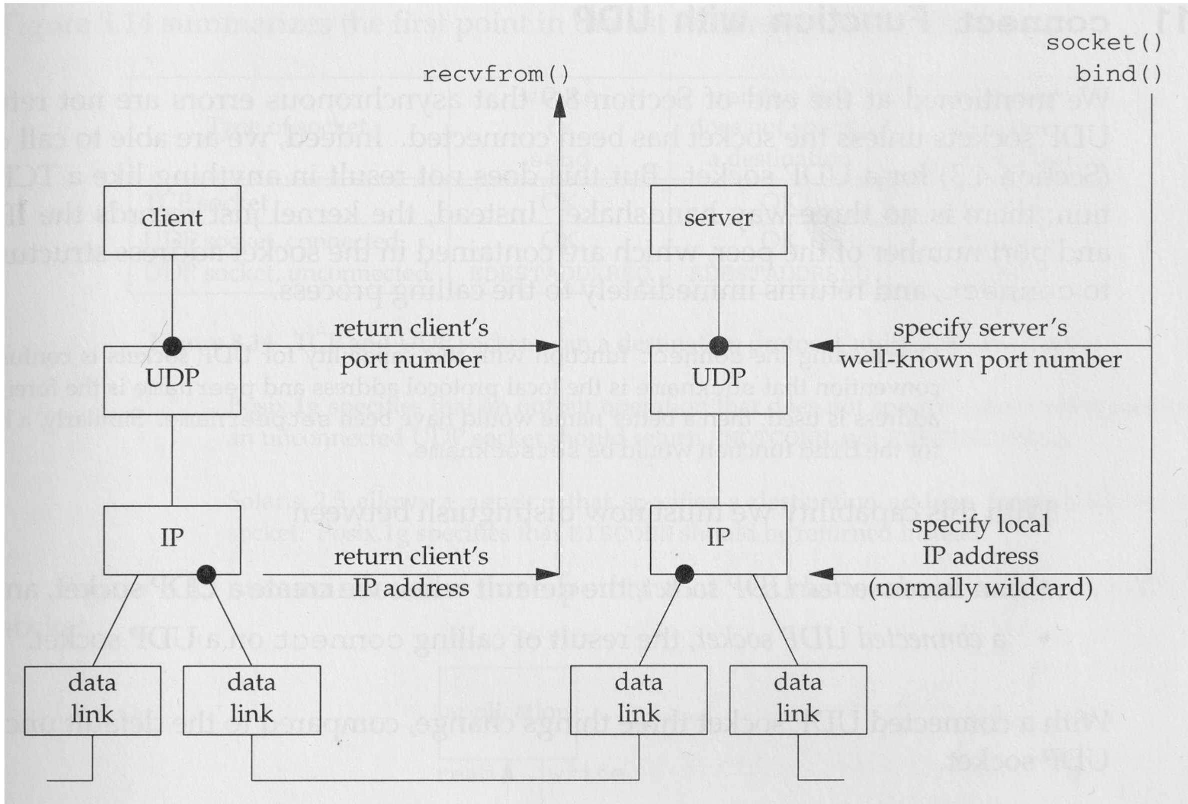


<그림 8.10> 클라이언트 관점에서의 UDP 클라이언트/서버의 요약

클라이언트는 sento 호출을 위해 반드시 서버의 IP address와 port 번호를 지정해야 한다. 우리는 클라이언트가 원하면 bind를 호출할 수 있다는 것을 언급하기는 했지만, 일반적으로, 클라이언트의 IP address와 port는 kernel에 의해 자동으로 결정된다. 만약 클라이언트를 위한 이 두 값들이 kernel에 의해 선택되면, 우리는 또한 클라이언트의 단말 port는 첫 번째 sendto에서 한 번 선택되고, 이 것을 절대로 변하지 않는다는 것을 언급했다.

클라이언트의 IP address는, 그렇지만, 클라이언트가 socket에 특정 IP address를 bind하지 않았다는 것을 가정하고, 클라이언트가 송신하는 모든 UDP datagram을 위해 변경될 수 있다. 그 이유는 상기 그림에서 볼 수 있다: 만약 클라이언트 host가 multihomed이면, 클라이언트는 두 목적지 사이를 교대해서, 한 쪽은 왼쪽의 datalink로 나가고, 다른쪽은 오른 쪽의 datalink로 나간다. 이 worst-case 시나리오에서, outgoing datalink 를 기반으로 해서 kernel에 의해 선택된 클라이언트의 IP address는, 모든 datagram을 위해 변경될 것이다.

다음 그림은 네 개의 값들을 보여주지만, 서버의 관점에서이다.



<그림 8.11> 서버 관점에서의 UDP 클라이언트/서버의 요약

From client's IP datagram	TCP server	UDP server
source IP address	accept	recvfrom
source port number	accept	recvfrom
destination IP address	getsockname	recvmsg
destination port number	getsockname	getsockname

<그림 8.12> 도착하는 IP datagram으로부터 서버가 이용 가능한 정보

TCP 서버는 연결된 socket을 위한 네 개의 정보 모두에 언제나 쉽게 접근할 수 있고, 이 네 개의 값들은 연결이 지속되는 동안은 상수로 남아있다. UDP socket에서는, 그렇지만, 목적지 IP address는 IPv4에서 IP_RECVDSTADDR socket 옵션이나 IPv6에서 IPV6_PKTINFO socket 옵션을 설정하고 recvfrom 대신 recvmsg를 호출해야만 얻을 수 있다.

8.7 UDP를 사용하는 CONNECT 함수

우리는 UDP socket이 connect되어 있지 않는 한 asynchronous error는 반환되지 않는다는 것을 언급했다. 실제로, 우리는 UDP socket을 위해 connect를 호출할 수 있다. 그렇지만 이것은 TCP 연결과 전혀 유사하지 않다: 여기에는 three-way handshake가 없다. 대신, kernel은 단지 immediate error들을 확인해서, 호출 process에게 즉시 반환한다.

UDP socket들을 위해 connect 함수에 이 능력을 추가하는 것은 혼란스럽다. 만약 sockname이 local protocol address이고 peername이 foreign address라는 convention이 사용된다면, 더 좋은 이름은 setpeername이 되었을 것이다. 비슷하게, bind 함수를 위한 더 좋은 이름은 setsockname 이 될 것이다.

이 기능과 함께, 우리는 이제 반드시 이들을 구별해야 한다:

- unconnected UDP socket, 우리가 UDP socket을 생성할 때 기본 socket
- connected UDP socket, UDP socket에서 connect를 호출했을 때의 결과

connected UDP socket은 기본적인 unconnected UDP socket에 비해 세 가지가 변경된다:

1. 우리는 출력 동작을 위해 목적지 IP address와 port를 더 이상 지정하지 않아도 된다. 이 말은, 우리는 sendto를 사용하지 않고, 대신 write 또는 send를 사용한다. connected UDP ocket으로 씌어지는 모든 것들은 connect에 의해 지정된 protocol address (e.g., IP address와 port) 로 송신된다.

   TCP와 유사하게, 우리는 connected UDP socket을 위해 sendto를 호출할 수 있지만, 우리는 목적지 address를 지정할 수 없다. sendto의 다섯 번째 argument (socket address structure로의 pointer)는 반드시 null pointer이어야 하고, 여섯 번째 argument (socket address structure의 크기) 는 0일 것이다. POSIX는 다섯 번째 argument가 null pointer이면, 여섯 번째 argument는 무시된다고 말하고 있다.

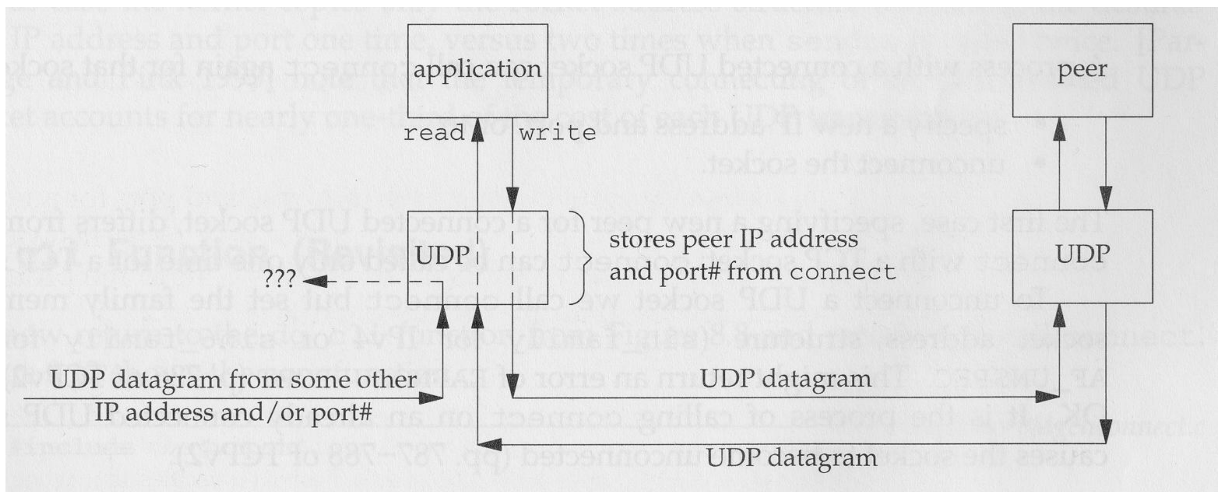
2. 우리는 datagram의 송신자에 대해 알기 위해 recvfrom의 사용할 필요가 없고, read, recv 또는 recvmsg를 대신 사용한다. connected UDP socket에서 입력 동작에 대해 kernel에 의해 반환되는 datagram들은 connect에서 지정된 protocol address에서 도착하는것들 뿐이다. connected UDP socket의 local protocol address (e.g., IP address 와 port) 가 목적지이지만 socket에 연결되지 않은 protocol address로부터 오는

datagram들은 connected socket으로 전달되지 않는다. 이것은 connected UDP socket이 반드시 한 peer와 datagram들을 교환하도록 제한한다.

multicast 또는 broadcast address에 connect하는 것이 가능하기 때문에, 기술적으로, connected UDP socket은 단지 하나의 IP address와 datagram들을 교환한다.

3. asynchronous error들은 connected UDP socket들을 위해 process로 반환된다. 결론은, 우리가 앞에서 기술한 바와 같이, unconnected UDP socket들은 asynchronous error들을 수신하지 못한다는 것이다.

다음 그림은 connected UDP socket에 대해 우리가 이야기한 세 가지 논점들을 요약한다.



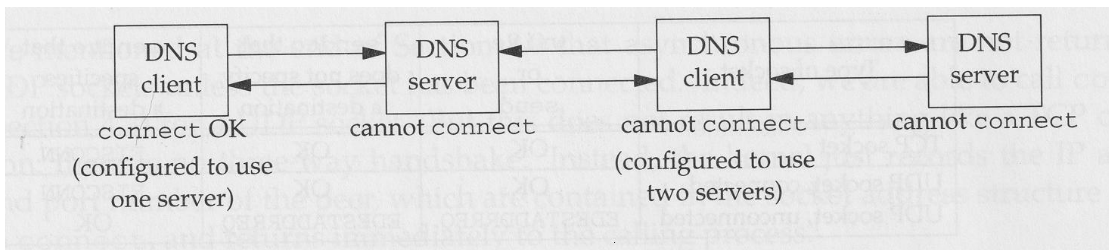
<그림 8.13> connect된 UDP socket.

application은 peer의 IP address와 port 번호를 지정해서, connect를 호출한다. application의 그 후 peer와 data를 교환하기 위해 read와 write를 사용한다.

다른 IP address 또는 port에서 오는 모든 datagram들은 IP address 또는 출발지 UDP port가 socket이 connect된 출발지 protocol address와 일치하지 않기 때문에 connected socket으로 전달되지 않는다. 이 datagram들은 host의 어떤 다른 UDP socket으로 전달되었을 것이다. 도착하는 datagram을 위해 일치하는 다른 socket이 없다면, UDP 는 이 datagram을 버리고 ICMP “port unreachable” error를 생성할 것이다.

요약하면, 우리는 UDP 클라이언트 또는 서버는 process가 UDP socket을 사용해서 정확히 하나의 peer와 통신할 때에만 connect를 호출할 수 있다고 말할 수 있다. 일반적으로, connect를 호출하는 것은 UDP 클라이언트이지만, UDP 서버가 단일 클라이언트와 오랜 시간 동안 통신하는 application들이 있다 (e.g., TFTP); 이 경우, 클라이언트와 서버 둘 다 connect를 호출할 수 있다.

DNS는 그림 8.14에서 보여지는 것처럼, 또 다른 예를 제공한다.



<그림 8.14> DNS 클라이언트와 서버와 connect 함수의 예제

DNS클라이언트는 일반적으로 /etc/resolv.conf 파일 내부에 서버들의 IP address를 나열함으로써, 하나 또는 그 이상의 서버들을 사용하도록 설정될 수 있다. 만약 하나의 서버가 기입되면 (figure에서 제일 왼쪽의 상자) 클라이언트는 connect를 호출할 수 있지만, 만약 여러 개의 서버들이 기입되면 (figure에서 오른쪽에서 두 번째 상자), 클라이언트는 connect를 호출할 수 없다. 또한, DNS 서버는 일반적으로 어떤 클라이언트의 요청이라도 처리하기 때문에, 서버는 connect를 호출할 수 없다.

UDP socket을 위해 connect를 여러 번 호출하기

connected UDP socket을 사용하는 process는 두 가지 이유들 중 하나로 그 socket을 위해 connect를 다시 호출할 수 있다:

- 새 IP address와 port를 지정하기 위해
- socket을 unconnect하기 위해

첫 번째 경우, connected UDP socket을 위해 새 peer를 지정하는 것은, TCP socket에서 connect를 사용하는 것과 구분된다: connect는 TCP socket을 위해 반드시 한 번만 호출될 수 있다.

UDP socket을 unconnect하기 위해, 우리는 connect를 호출하지만 socket address structure의 family member (IPv4에서 sin_family 또는 IPv6에서 sin6_family)를 AF_UNSPEC으로 설정한다. EAFNOSUPPORT error가 반환될 수도 있지만, 이것은 받아들일 수 있다. socket이 unconnected가 되도록 이미 연결된 UDP socket에 connect를 호출하는 것은 process이다.

Unix 변종(variant)들은 정확히 어떻게 socket을 unconnect하는 지에 대해서는 서로 방식을 달리 하는 것으로 보이고, 여러분은 어떤 system들에서는 동작하지만 다른 system에서는 동직하지 않는 접근 방식들을 보게 될 수도 있다. 예를 들어, address를 NULL 로 해서 connect를 호출하는 것은 몇몇 system들에서만 동작한다 (그리고, 이것은 세 번째 argument, length, 가 0이 아닐 때만 동작한다.). POSIX specification과 BSD man page들은 단지 null address만이 사용되어야 한다고 언급하고 있고 반환하는 error에 대해서는 언급하고 있지 않아서(성공인 경우에도), 여기에서는 별로 도움이 되지 않는다. 가장 portable한 해결책은 address structure를 0으로 하고, 위에서 언급했던 것처럼 family를 AF_UNSPEC으로 설정해서, 이것을 connect에게 전달하는 것이다.

성능

application이 unconnected UDP socket에서 sendto를 호출하면, Berkeley-derived kernel들은 일시적으로 socket을 connect하고, datagram을 송신하고, 그 후 socket을 unconnect한다. unconnected UDP socket에서 두 datagram들을 위해 sendto를 송신하는 것은 kernel에 의해 다음의 여섯 단계를 포함한다:

- socket을 connect한다
- 첫 번째 datagram을 출력한다
- socket을 unconnect한다
- socket을 connect한다
- 두 번째 datagram을 출력한다
- socket을 unconnect한다

application이 동일한 peer로 여러 개의 datagram들을 송신하게 될 것이라는 것을 알 때는, socket을 명시적으로 connect하는 것이 더 효율적이다. connect를 호출하고 write를 두 번 호출하는 것은 kenrel에 의해 다음의 단계들을 포함한다:

- socket을 connect한다.
- 첫 번째 datagram을 출력한다
- 두 번째 datagram을 출력한다

이 경우, kernel은 목적지 IP address와 port를 포함하고 있는 socket address structure를 단지 한 번만 복사하고, 그에 비해 sendto가 두 번 호출되었을 때는 두 번 복사된다. [Partridge and Pink 1993]은 unconnected UDP socket을 일시적으로 connet하는 것으로 각각의 UDP 전송에서 3분의 1의 비용을 기대할 수 있다고 적고 있다.

dg_cli 함수 (Revisited)

우리는 이제 dg_cli() 함수로 돌아가서 connect를 호출하도록 다시 코딩한다. 코드는 새 함수를 보여준다.

udpcliserv/dgcliconnect.c

```
 1 #include  "unp.h"

 2 void
 3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 4 {
 5     int     n;
 6     char    sendline[MAXLINE], recvline[MAXLINE + 1];

 7     Connect(sockfd, (SA *) pservaddr, servlen);

 8     while (Fgets(sendline, MAXLINE, fp) != NULL) {

 9         Write(sockfd, sendline, strlen(sendline));

10         n = Read(sockfd, recvline, MAXLINE);

11         recvline[n] = 0;        /* null terminate */
12         Fputs(recvline, stdout);
13     }
14 }
```

udpcliserv/dgcliconnect.c

<그림 8.15> connect를 호출하는 dg_cli 함수.

변경된 사항은 새로운 connect 호출과 sendto와 recvfrom 호출을 write와 read 호출로 교체하는 것이다. 이 함수는 connect로 전달되는 socket address structure의 내부를 보지 않는 관계로 여전히 protocol-independent이다. 우리의 클라이언트 main 함수, Figure 8.7은, 원래 형태를 유지한다.

만약 우리가 host macosx에서 이 프로그램을 실행시키고, host freebsd4 (우리의 서버를 port 9877에서 실행시키고 있지 않은) 의 IP address를 지정하면, 우리는 다음의 출력을 얻는다:

```
maxosx % udpcli04 172.24.37.94
hello, world
read error: Connection refused
```

우리가 인식할 수 있는 첫 번째 점은 우리는 클라이언트 process를 시작할 때 error를 받지 못한다는 것이다. error는 우리가 첫 번째 datagram을 서버로 송신한 뒤에야 발생한다. 이 datagram을 송신함으로써 ICMP error를 서버 host로부터 이끌어내게 된다. 그렇지만 TCP 클라이언트가 서버 process를 실행시키지 않는 서버 host를 지정해서 connect를 호출하면, connect 호출은 TCP three-way handshake가 발생되게 하고, 그 handshake의 첫 번째 packet은 서버 TCP로부터 RST를 이끌어내기 때문에, connect는 error를 반환한다.

다음은 tcpdump 출력을 보여준다.

```
macosx % tcpdump
1  0.0        macosx.51139 > freebsd4.9877: udp 13
2  0.006180   freebsd4 > macosx: icmp: freebsd4 udp port 9877 unreachable
```

ICMP error는 kernel에 의해 error ECONNREFUSED로 매핑되고, 이 error는 우리의 err_sys 함수의 message string 출력 “Connection refused”에 연관된다.

8.8 FLOW CONTROL이 없는 UDP

우리는 이제 flow control을 가지고 있지 않은 UDP의 효과에 대해 살펴볼 것이다. 첫 번째로, 우리는 dg_cli() 함수를 정해진 수의 datagram들을 송신하도록 수정한다. 이 함수는 더 이상 standard input으로부터 읽지 않는다. 다음 코드는 새 버전을 보여주고 있다. 이 함수는 서버로 2,000개의 1,400-byte짜리 UDP datagram들의 write한다.

udpcliserv/dgcliloop1.c

```
1 #include    "unp.h"

2 #define NDG     2000        /* #datagrams to send */
3 #define DGLEN   1400        /* length of each datagram */

4 void
5 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
6 {
7     int     i;
8     char    sendline[DGLEN];

9     for (i = 0; i < NDG; i++) {
10        Sendto(sockfd, sendline, DGLEN, 0, pservaddr, servlen);
11    }
12 }
```

udpcliserv/dgcliloop1.c

<그림 8.16> 고정된 수의 datagram을 서버로 write하는 dg_cli 함수

udpcliserv/dgecholoop1.c

```
1 #include    "unp.h"

2 static void recvfrom_int(int);
3 static int count;

4 void
5 dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
6 {
7     socklen_t len;
8     char    mesg[MAXLINE];

9     Signal(SIGINT, recvfrom_int);

10    for ( ; ; ) {
11        len = clilen;
12        Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);

13        count++;
14    }
15 }

16 static void
17 recvfrom_int(int signo)
18 {
19    printf("\nreceived %d datagrams\n", count);
20    exit(0);
21 }
```

udpcliserv/dgecholoop1.c

<그림 8.17> 수신 받은 datagram들을 계산하는 dg_echo 함수

다음으로 datagram들을 수신하고 받은 datagram들의 개수를 세도록 서버를 변경한다. 이 서버는 더 이상 클라이언트로 datagram들을 echo해서 돌려보내지 않는다. 그림 8.17은 새 dg_echo 함수를 보여준다. 우리가 terminal interrupt key (SIGINT)로 서버를 종료하면, 이 함수는 수신 받은 datagram들이 수를 출력하고 종료한다.

이제 느린 SPARCStation인, host freebsd에서 서버를 실행시킬 것이다. 우리는 100Mbps Ethernet에 직접 연결되어 있는, RS/6000 system인 aix에서 클라이언트를 실행시킨다. 추가적으로, 우리는 얼마나 많은 datagram들이 사라졌는지를 알려주는 통계로, 실행 전과 후에, 서버에서 netstat –s를 실행시킨다.

클라이언트는 2.000개의 datagram들을 송신했지만, 서버 application은 이들 중 30개만 수신했고, 98%의 loss rate이다. 이 datagram들이 서버 application에서 사라졌는지 또는 클라이언트 application에서 사라졌는지 알려주는 것은 없다. 우리가 말했듯이, UDP는 flow control을 가지지 않고, unreliable하다.

netstat 출력을 보면, 서버 host(서버 application이 아닌)가 수신 받은 datagram들의 총 개수는 2,000 (73,208 - 71,208) 이다. "dropped due to full socket buffers" 카운터는 얼마나 많은 datagram들이 UDP에 의해 수신되었고 수신 받는 socket 의 receive queue가 가득 차서 버려졌는지를 나타낸다. 이 값은 1970 (3,491 – 1,971)이고, application의 카운터 출력(30)과 더해지면, host가 수신 받은 datagram 2000개와 동일하다. 유감스럽게도, netstat이 계산하는 socket buffer가 가득 차서 버려진 수는 system-wide이다. 어떤 application들 (e.g., 어떤 UDP port들) 이 영향을 받았는지 판단할 방법은 없다.

이 예에서 서버에 의해 수신된 datagram들의 수는 예견할 수 없다. 이것은 network load, 클라이언트 host의 process load, 그리고 서버의 processing load 같은 요인에 따라 달라진다.

만약 우리가 동일한 클라이언트와 서버를 실행시키지만, 이번에는 클라이언트를 느린 Sun에서 실행시키고 서버를 빠른 RS/6000에서 실행시키면, 어떤 datagram들도 버려지지 않는다.

```
aix % udpserv06
^?                          우리는 클라이언트가 종료된 후 interrupt key를 입력한다
received 2000 datagrams
```

UDP Socket Receive Buffer

주어진 socket에 대해 UDP 에 의해 queue되는 UDP datagram들의 수는 그 socket의 receive buffer 크기에 의해 제한된다. 우리는 SO_RCVBUF socket 옵션으로 이것을 변경할 수 있다. FreeBSD에서 UDP socket receive buffer의 기본 크기는 42.080 byte여서, 우리의 1,400-byte datagram들이 30개밖에 들어갈 수 없다. 만약 우리가 socket receive buffer의 크기를 증가시키면, 우리는 서버가 추가적인 datagram들을 수신할 것을 기대할 수 있다.

다음 dg_echo 함수에서 socket receive buffer를 240KB로 변경하는 것을 보여준다.

udpcliserv/dgecholoop2.c

```
1 #include "unp.h"

2 static void recvfrom_int(int);
3 static int count;

4 void
5 dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
6 {
7     int     n;
8     socklen_t len;
9     char    mesg[MAXLINE];

10    Signal(SIGINT, recvfrom_int);

11    n = 240 * 1024;
12    Setsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &n, sizeof(n));

13    for ( ; ; ) {
14        len = clilen;
15        Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);

16        count++;
17    }
18 }

19 static void
20 recvfrom_int(int signo)
21 {
22    printf("\nreceived %d datagrams\n", count);
23    exit(0);
24 }
```

udpcliserv/dgecholoop2.c

<그림 8.18> socket receive queue의 크기를 증가시키는 dg_echo 함수.

만약 우리가 이 서버를 Sun에서 실행시키고 클라이언트를 RS/6000에서 실행시키면, 수신 받은 datagram들의 수는 이제 103이다. 기본 socket receive buffer를 가진 이전의 예보다는 약간 낫지만, 이것이 만병 통치약은 아니다.

8.9 UDP에서 OUTGOING INTERFACE의 결정

connected UDP socket은 또는 특정 목적지에 사용될 outgoing interface를 판단하는 데 사용될 수 있다. 이것은 UDP socket이 connect 함수에 적용되었을 때의 부작용이다: kernel이 local IP address를 선택한다 (명시적으로 이것을 할당하기 위해 process가 bind를 호출하지 않았을 경우). 이 local IP address는 목적지 IP address까지의 routing table을 검색하고, 그 후 결과로 얻은 interface를 위한 primary IP address를 사용함으로써 선택된다.

다음 그림은 특정 IP address로 connect해서 그 후 getsockname을 호출하여, local IP address와 port를 출력하는 간단한 UDP 프로그램을 보여준다.

udpcliserv/udpcli09.c

```
 1 #include     "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     sockfd;
 6     socklen_t len;
 7     struct sockaddr_in cliaddr, servaddr;

 8     if (argc != 2)
 9         err_quit("usage: udpcli <IPaddress>");

10     sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

11     bzero(&servaddr, sizeof(servaddr));
12     servaddr.sin_family = AF_INET;
13     servaddr.sin_port = htons(SERV_PORT);
14     Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

15     Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));

16     len = sizeof(cliaddr);
17     Getsockname(sockfd, (SA *) &cliaddr, &len);
18     printf("local address %s\n", Sock_ntop((SA *) &cliaddr, len));

19     exit(0);
20 }
```

udpcliserv/udpcli09.c

<그림 8.19> outgoing interface를 확인하기 위해 connect를 사용하는 UDP 프로그램.

만약 우리가 multihomed host freebsd에서 프로그램을 실행시키면, 다음의 출력을 얻는다:

```
freebsd % udpcli09 206.168.112.96
local address 12.106.32.254:52329
freebsd % udpcli09 192.168.42.2
local address 192.168.42.2:52330
freebsd % udpcli09 127.0.0.1
local address 127.0.0.1:52331
```

9. SCTP SOCKETS

SCTP는 새로운 수송계층 프로토콜로써 2000년에 IETF에서 표준화되었다. (TCP는 1981년에 표준화되었다.) SCTP는 성장하는 IP 전화통신 시장의 수요에 의해 디자인되었고, 특히 인터넷을 통한 전화서비스 신호 전달을 목적으로 한다. 디자인을 이행하기 위한 요구사항은 RFC2719에 기술되어있다. SCTP는 신뢰성 있고, 메시지 지향성(message-oriented) 프로토콜이며, 단말(endpoint) 사이에 다중스트림(multi-stream)을 제공하고 수송계층에서 멀티홈잉(multi-homing)을 지원한다. SCTP가 새로운 프로토콜로 제정된 이후 TCP나 UDP처럼 널리 사용되지는 않았다. 그러나 SCTP가 제공하는 몇 가지 새로운 특성은 응용 디자인을 확실히 단순화 시킬지도 모른다.

비록 SCTP와 TCP 사이에는 몇 가지 근본적인 차이점을 가지고 있으나, SCTP의 일대일(one-to-one) 인터페이스에서는 TCP와 매우 유사한 응용 인터페이스를 제공한다. 이것은 응용의 사소한 변경을 요구하지만 SCTP의 몇 가지 추가된 특성의 사용은 허락하지 않는다. SCTP의 일대다(one-to-many) 인터페이스는 SCTP의 특성을 모두 지원한다. 그러나 이것은 기존의 응용의 많은 부분을 변경해야 할지도 모른다. 따라서 일대다(one-to-many) 인터페이스는 SCTP로 개발되는 새로운 응용에 권고된다.

이 장에서는 SCTP에서 사용될 수 있는 추가적인 기본 소켓 함수에 대해서 기술한다. 우리는 먼저 응용 개발자들에 의해 사용될 수 있는 두 가지 다른 인터페이스 모델에 대해 기술한다. 10장에서는 일대다(one-to-many) 모델에서 사용하는 에코 서버(echo server)를 구현해 볼 것이다. 또한 SCTP에서만 사용되는 새로운 함수를 기술한다. 우리는 shutdown 함수가 SCTP에서 TCP와는 어떻게 다르게 사용하는지도 살펴본다. 그리고 나서 SCTP의 통지(notifications)의 사용을 간단히 살펴본다. 통지는 사용자 데이터의 도착과는 다른, 프로토콜에서 중요한 이벤트를 응용에게 알려주는 역할을 한다.

SCTP는 새로운 프로토콜이 된 이후, SCTP의 모든 특성을 위한 인터페이스는 아직 완전히 안정화되지 않았다. 이 글에서는 인터페이스가 안정화 되었다고 믿고 기술한다. 하지만 아직 소켓 API처럼 널리 퍼지지는 않았다. SCTP의 사용을 기반으로 디자인된 응용의 사용자들은 커널 패치를 준비하거나 운영체제의 업그레이드를 필요로 할지도 모른다. 그리고 그 응용이 널리 쓰여야 하는 프로그램이라면 SCTP가 지원되지 않는 시스템을 고려하여 TCP를 사용할 수 있도록 해야 한다.

9.1 INTERFACE 모델

SCTP에는 일대일(one-to-one) 소켓 방식과 일대다(one-to-many) 소켓 방식 두 가지가 있다. 일대일(one-to-one) 소켓은 정확히 하나의 SCTP association과 대응한다. SCTP의 일대일(one-to-one) 소켓과 SCTP association 사이의 관계는 TCP 소켓과 TCP 연결 사이의 관계와 유사하다. 일대다(one-to-many) 소켓에서는, 여러 개의 SCTP association은 주어진 소켓을 동시에 활성화할 수 있다. SCTP 일대다(one-to-many) 소켓과 SCTP association의 관계는 특정 포트로 바인드 되어 모두 동시에 데이터를 전송하는 여러 원격 UDP 단말로부터 interleaved datagram을 받을 수 있는 UDP 소켓과 유사하다.

사용하는 인터페이스의 스타일을 결정하고자 할 때, 응용은 다음 요소를 고려할 필요가 있다.

- 작성하는 서버의 방식은 순차접속(iterative) 방식인가? 다중접속(concurrent) 방식인가?
- 서버는 얼마나 많은 소켓 디스크립터를 관리하는가?
- Four-way handshake의 3번째와 가능하다면 4번째 패킷 상에 데이터를 함께 전송할 수 있도록 association 설정을 최적화하는 것이 중요한가?
- 응용에서는 얼마나 많은 연결 상태를 유지해야 하는가?

SCTP의 소켓 API로 개발할 때, 문서나 소스 코드에서 여기서 언급한 소켓의 두 가지 방식에 대한 용어가 아닌 다른 용어를 볼 수 있을지도 모른다. 일대일 소켓의 초기 용어는 "TCP-style" 소켓이고, 일대다 소켓의 초기 용어는 "UDP-style" 소켓이다.

소켓 스타일의 사용에 의존하여 SCTP가 TCP나 UDP처럼 동작할 것이라고 생각하게 함으로써 혼란을 가져올 수 있기 때문에 이 스타일 용어는 이후에 사라질 것이다. 실제로, 이 용어들은 단지 TCP와 UDP 소켓 사이에서 하나의 측면만 언급한다. (즉, 하나의 소켓이 다중 동시 수송계층 association을 제공할지도 모른다.) 현재의 용어("one-to-one" versus "one-to-many")는 두 소켓 스타일 사이에서 차이점에 초점을 맞추어 사용하고 있다. 그리고 일대다(one-to-many)이란 용어 대신에 다대일(many-to-one)이란 용어로 바꾸어 사용할 수도 있다.

One-to-One Style 소켓

일대일(one-to-one) 스타일은 기존에 존재하는 TCP 응용에 변경을 가하면 SCTP 응용으로 쉽게 개발할 수 있다. 그것은 4장에서 기술한 identical 모델과 유사하다. 기존의 TCP 응용을 SCTP 응용으로 변경하기 위해서 아래와 같은 차이점을 고려해야 한다.

1. 어떤 소켓 옵션이라도 동등하게 TCP에서 SCTP로 변환될 수 있다. 연결 설정시 일반적인 옵션은 TCP_NODELAY와 TCP_MAXSEG이다. 이것은 SCTP_NODELAY와 SCTP_MAXSEG로 쉽게 대응시킬 수 있다.

2. SCTP는 메시지 경계를 유지하므로 응용계층 메시지 bound는 요구하지 않는다. 예를 들어 TCP 기반의 응용 프로토콜은 write() 시스템 콜을 사용하여 2바이트 메시지 길이 필드를 쓰고 잇달아 write() 시스템 콜이 x바이트 데이터를 쓴다. 그러나 SCTP에서는 수신하는 SCTP는 두 개의 별도의 메시지로 수신할 것이다. 즉, read() 시스템 콜은 두 번 호출될 것이다. 한번은 2바이트 메시지를 수신하고, 그리고 나서 다시 x바이트 메시지를 수신할 것이다.

3. 어떤 TCP 응용은 통신하는 상대방에게 입력의 끝이라는 신호를 보냄으로써 half-close를 사용한다. 이 특성을 가진 TCP 응용을 SCTP로 이식하기 위해서, 응용계층 프로토콜은 응용 데이터 스트림 안에서 입력의 끝이라는 응용 신호를 위해 다시 작성해야 할 것이다.

4. send 함수는 일반적인 상황에서 사용될 수 있다. sendto와 sendmsg 함수에서는 주소 정보에 우선 종착 주소(primary destination address)의 정보를 넣는 것을 포함한다.

그림 9.1은 일대일(one-to-one) 스타일의 시간 흐름을 보여준다. 서버가 시작하면 소켓을 열고, 주소를 바인드하고, accept 시스템 콜을 사용하여 클라이언트의 연결을 기다린다. 기다리고 있으면 그 뒤 클라이언트는 통신을 하기 위해 시작하면 소켓을 열고, 서버와의 association을 초기화한다. 클라이언트는 서버에게 요청을 전송하고, 서버는 그 요청을 처리하고 클라이언트에게 요청에 대한 응답을 전송하는 것을 가정한다. 이 주기(cycle)는 클라이언트가 맺은 association에서 shutdown을 시작할 때까지 계속된다. shutdown으로 association은 끊어지고, 서버는 종료하거나 새로운 association을 기다린다. 전형적인 TCP와 비교하면 SCTP 일대일 소켓 스타일은 그림 4.1에서 보여지는 것처럼 매우 유사하게 처리된다.

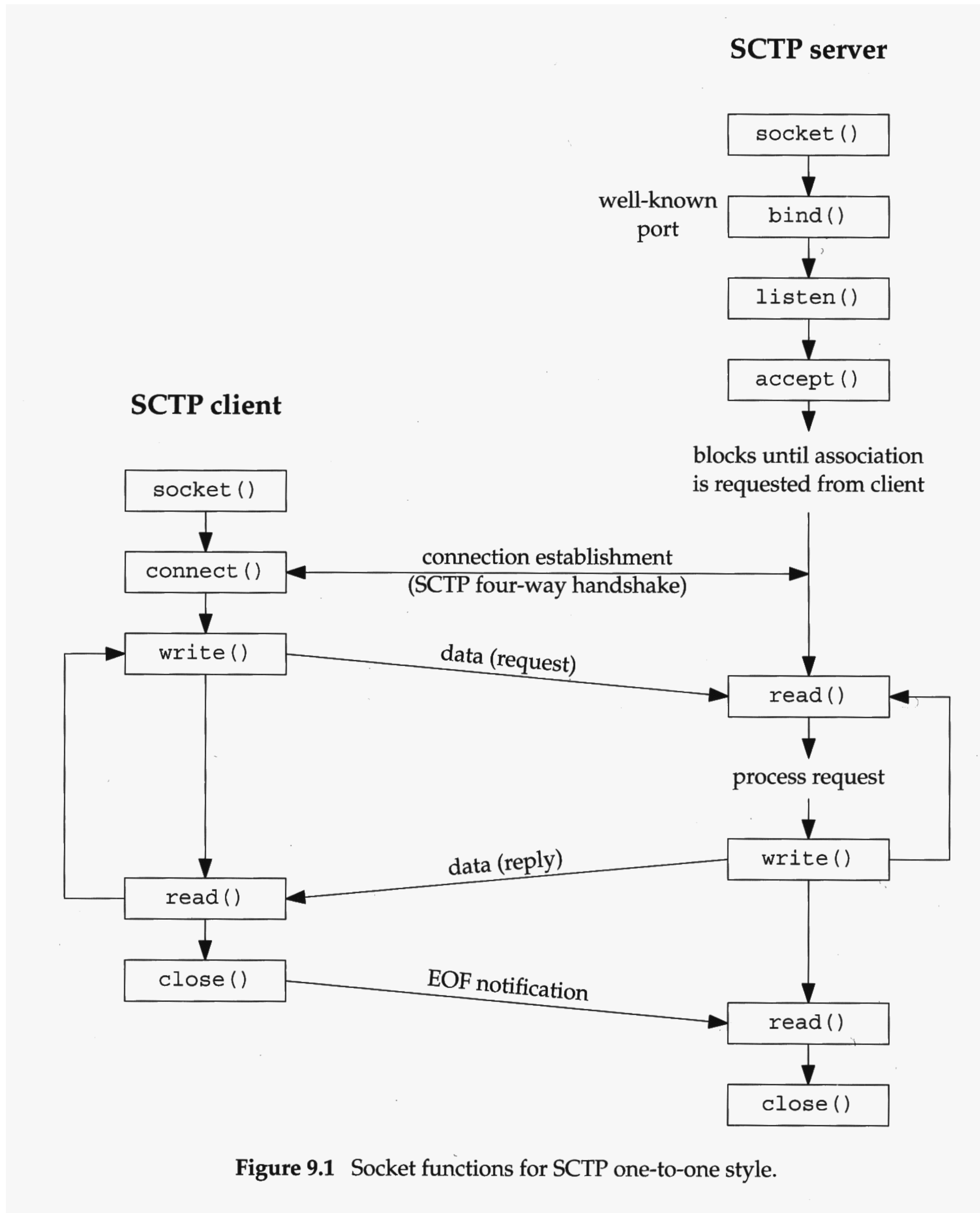


Figure 9.1 Socket functions for SCTP one-to-one style.

일대일(one-to-one) 스타일의 SCTP 소켓을 생성할 경우, IP socket에서 *family* 인자는 AF_INET 또는 AF_INET6로, *type* 인자는 SOCK_STREAM으로 *protocol* 인자는 IPPROTO_SCTP로 설정한다.

One-to-Many Style 소켓

일대다(one-to-many) 스타일은 많은 소켓 디스크립터의 관리 없이 서버에게 응용 사용자가 쓸 수 있도록 한다. 하나의 소켓 디스크립터는 다중 association을 나타낼 것이고, UDP 소켓이 다중 클라이언트로부터 메시지를 수신 받을 수 있는 방법과 동일하다. Association 식별자는 일대다(one-to-many) 스타일 소켓상에서 단일의 association으로 식별할 수 있도록 사용된다. 이 association 식별자는 sctp_assoc_t의 값이다. 일반적으로 정수이며 정해지지 않은 값이다. 응용은 커널에 의해 미리 주어진 association 식별자만 사용해야 하며 주어지지 않은 association 식별자를 사용해서는 안 된다. 일대다(one-to-many) 스타일의 사용자는 다음의 내용을 지켜야만 한다.

1. 클라이언트가 association을 종료할 경우 서버 측에서는 자동적으로 종료될 것이고 커널 내부에 association을 위한 상태도 모두 삭제되어야 한다.

2. 일대다(one-to-many) 스타일의 사용은 four-way handshake의 3, 4번째 패킷에 데이터를 piggyback 할 수 있는 유일한 방법이다.

3. sendto, sendmsg 또는 sctp_sendmsg 함수는 아직 association이 존재하지 않는 주소를 위해 active open을 시도할 것이고, 이 시도가 성공한다면 그 주소에 새로운 association을 생성할 수 있다. 비록 응용이 passive open을 요구하는 listen 함수를 호출하여 전송을 하더라도 이 동작은 수행된다.

4. 사용자는 sendto, sendmsg 또는 sctp_sendmsg 함수를 사용해야만 하고 send나 write 함수는 사용하지 않을 것이다. 만약 일대일(one-to-one) 소켓을 생성할 때 sctp_peeloff 함수를 사용한다면 send나 write 함수를 사용할 수도 있다.

5. 언제라도 send 함수 중 하나를 호출하면 MSG_ADDR_OVER 플래그의 설정이 없다면 association 초기화 시간에 시스템에 의해 선택된 우선 종착 주소(primary destination address)를 sctp_sndrcvinfo 구조체에서 사용될 것이다. 이를 위해 보조적인 데이터와 함께 sendmsg 함수를 사용하거나 sctp_sendmsg 함수 사용을 필요로 할 것이다.

6. 9절에서 다룰 SCTP 통지 중 하나인 association 이벤트 사용이 가능하다. 만약 응용이 이 이벤트를 수신 받는 것을 원하지 않는다면 SCTP_EVENTS 소켓 옵션을 이용해 명시적으로 이벤트를 수신 받지 않도록 한다. 기본적으로 사용 가능한 이벤트는 sctp_data_io_event이고 이 이벤트는 recvmsg와 sctp_recvmsg 함수에서 보조적인 데이터를 제공한다. 이벤트에 대한 기본 설정은 일대일(one-to-one) 스타일과 일대다(one-to-many) 스타일 모두 적용된다.

SCTP 소켓 API가 처음 개발되었을 때, 일대다(one-to-many) 스타일 인터페이스는 기본 설정에 의해서만 association 통지가 정의되었다. 그 이후 API 문서에서는 일대일(one-to-one)과 일대다(one-to-many) 스타일에서 sctp_data_io_event를 제외한 모든 통지를 사용하지 않도록 할 수 있다. 그러나 모든 구현에서 이렇게 동작이 이루어지는 것은 아니다. 이것처럼 원하거나 원하지 않는 통지를 명시적으로 실행 가능하게 하거나 가능하지 않게 하는 것은 응용 사용자에게 좋은 방법이다. 이 명시적 접근 방법은 개발자가 운영체제 코드를 이식하는데 있어 예측한 동작에 대한 결과를 보장한다.

일대다(one-to-many) 스타일의 일반적인 시간 흐름은 그림 9.2에서 보여주고 있다. 먼저 서버가 시작되면 소켓이 생성되고 주소를 바인드하며 클라이언트의 association을 가능하게 하기 위해 listen 함수를 호출한다. 그 후 sctp_recvmsg 함수를 호출하는데 이 함수는 첫 번째 메시지가 도착할 때까지 기다리며 block 하고 있다. 클라이언트는 소켓을 열고 sctp_sendto 함수를 호출하는데 이 함수는 내부적으로 서버와 association을 설정하고 four-way handshake의 세 번째 패킷상에서 서버에게 데이터 요청을 piggyback 한다. 서버는 클라이언트의 요청을 수신하여 처리하고 다시 그 응답을 전송한다. 클라이언트는 응답을 수신 받고 소켓을 종료한 후 association을 끊는다. 서버는 다음 메시지를 수신 받을 때까지 loopback을 수행한다.

위의 예제는 많은 association(클라이언트)으로부터의 메시지를 단일 쓰레드의 제어로 처리할 수 있는 순차접속(iterative) 방식의 서버를 보여준다. SCTP에서 일대다 소켓은 sctp_peeloff 함수를 사용하여 순차접속(iterative)와 다중접속(concurrent) 서버 모델이 조합된 서버 형태로 사용할 수 있고 그 특징은 아래와 같다.

1. sctp_peeloff 함수는 오래 지속되는 세션의 경우 일대다(one-to-many) 소켓으로부터 일대일(one-to-one) 소켓으로 된 하나의 association으로 분리할 수 있다.
2. 1분리된 association의 일대일 소켓은 다중접속(concurrent) 모델에서처럼 쓰레드나 프로세스를 이용하여 자원을 할당할 수 있다.
3. 그 동안, 메인 쓰레드는 기존 일대다(one-to-many) 소켓상의 순차접속(iterative)안에서 다른 association부터의 메시지를 처리하는 작업을 계속 수행한다.

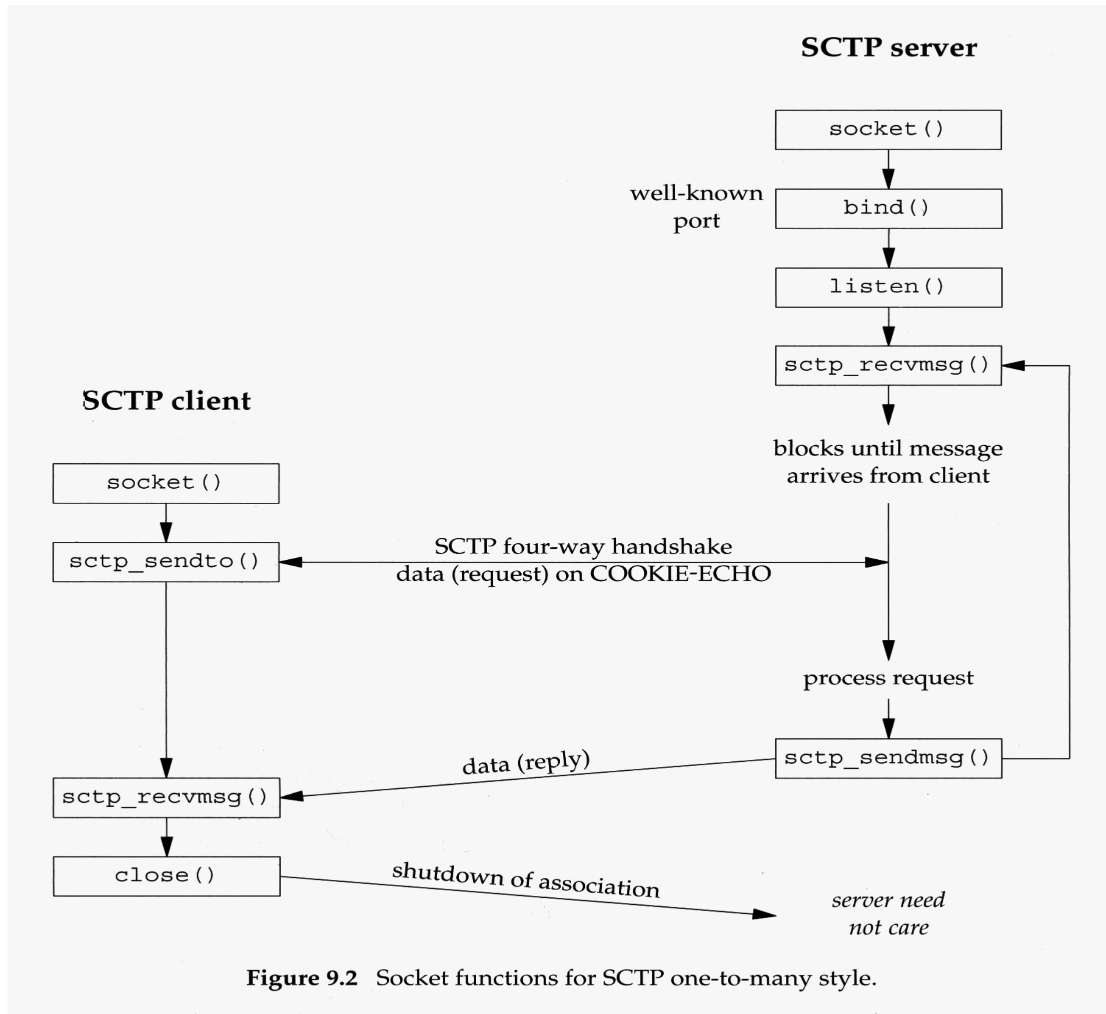


Figure 9.2 Socket functions for SCTP one-to-many style.

one-to-many 스타일의 SCTP 소켓을 생성할 경우, IP socket에서 *family* 인자는 AF_INET 또는 AF_INET6로, *type* 인자는 SOCK_SEQPACKET으로 *protocol* 인자는 IPPROTO_SCTP로 설정한다.

9.2 SCTP_BINDX 및 SCTP_BINDX 함수

sctp_bindx 함수

SCTP 서버는 호스트 시스템의 IP 주소들 중의 일부분만 바인드 하기를 원할 수도 있다. 전통적으로 TCP나 UDP 서버는 호스트 상의 하나의 IP 주소나 전체의 IP 주소들을 바인드 할 수는 있지만 IP 주소들 중의 일부분을 바인드 할 수 없다. sctp_bindx 함수는 SCTP 소켓이 IP 주소들 중의 일부분을 바인드 할 수 있도록 함으로써 TCP나 UDP에 비해 유연함을 제공한다.

```
#include <netinet/sctp.h>

int sctp_bindx( int sockfd, const struct sockaddr *addrs, int addrcnt,
int flags );

                                         Returns: 0 if OK, -1 on error
```

*sockfd*는 소켓 디스크립터로써 socket 함수에 의해 반환된다. 두 번째 매개변수인 *addrs*는 바인드 할 IP 주소들 목록의 포인터이다. 각 소켓 주소 구조체는 즉시 버퍼에 위치하고 패딩(padding) 없이 이전의 소켓 주소 구조체 뒤에 위치한다. 그림 9.4는 이것의 한 예이다.

sctp_bindx에 사용되는 주소들의 수는 *addrcnt* 매개변수에 의해 명시된다. *flag* 매개변수는 그림 9.3의 두 가지 동작 중의 하나를 수행하도록 sctp_bindx 콜에 지시한다.

flags	Description
SCTP_BINDX_ADD_ADDR	Add the address(es) to the socket
SCTP_BINDX_REM_ADDR	Remove the address(es) from the socket

Figure 9.3 *flags* used with sctp_bindx function.

sctp_bindx 콜은 bound가 있거나 bound가 없는 소켓에 사용할 수 있다. bound가 없는 소켓에서의 sctp_bindx는 소켓 디스크립터에서 주어진 주소들을 바인드 할 것이다. sctp_bindx를 bound 소켓에서 사용한다면, 소켓 디스크립터에 추가적인 주소들을 연결하기 위해 SCTP_BINDX_ADD_ADDR을 사용할 수 있고 소켓 디스크립터에 이미 연결되어 있는 주소들을 제거하기 위해 SCTP_BINDX_REM_ADDR을 사용할 수 있다. sctp_bindx가 listening 소켓상에서 수행된다면 후의 association은 새로운 주소 배치를 할 것이고, 그 변화는 이전의 존재하는 association들에는 영향을 미치지 않는다. sctp_bindx의 플래그의 두 값들은 상호 배타적이므로, 둘 다 설정한다면 sctp_bindx는 실패할 것이고 오류 코드 EINVAL을 반환한다. 모든 소켓 주소 구조체 안의 포트 번호는 같아야만 하고 이미 bound된 포트 번호는 대응되어 있어야만 한다. 그러지 않으면 sctp_bindx는 실패할 것이고 오류 코드 EINVAL을 반환한다.

만약 단말이 동적 주소 특성을 제공한다면 sctp_bindx에서의 SCTP_BINDX_ADD_ADDR이나 SCTP_BINDX_REM_ADDR 플래그는 단말이 상대방 단말의 주소 목록을 변경하기 위해 상대방 단말에게 적당한 메시지를 전송하도록 하게 할 것이다. 연결된 association으로부터 주소의 추가 및 삭제는 부가적인 특성이므로 이 특성을 지원하지 않는 구현에서는 EOPNOTSUPP를 반환할 것이

다. Association의 양 끝에서는 적당한 연산을 위해 이 특성을 지원해야만 한다. 인터페이스의 동적 공급을 지원하는 시스템이라면 이 특성은 유용하게 사용할 수 있다. 예를 들어, 새로운 이더넷 인터페이스를 가져온다면 응용은 기존의 연결상에서 추가적인 인터페이스의 사용을 시작하기 위해 SCTP_BINDX_ADD_ADDR을 사용할 수 있다.

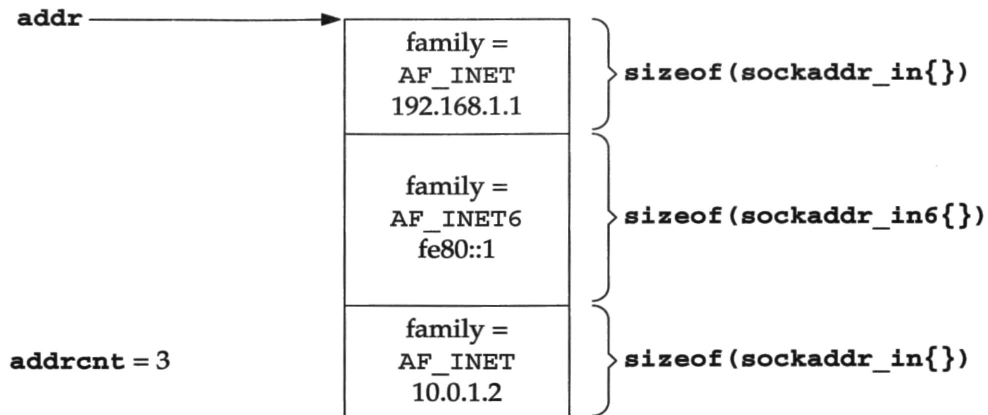


Figure 9.4 Packed address list format for SCTP calls.

sctp_connectx 함수

```
#include <netinet/sctp.h>

int sctp_connectx( int sockfd, const struct sockaddr *addrs, int
addrcnt );

                                    Returns: 0 for success, -1 on error
```

sctp_connectx 함수는 멀티홈잉을 지원하는 상대방 단말에 연결할 때 사용한다. 한 상대방 단말이 소유하고 있는 IP 주소들의 수는 *addrcnt*로 명시한다. *addrs* 매개변수는 그림 9.4와 같이 IP 주소들 목록의 point를 가리킨다. SCTP 스택은 association을 설립하기 위해 주어진 하나 또는 그 이상의 IP 주소들에 사용한다. *addrs*에 목록화된 모든 IP 주소들은 유효하고 인증된 IP 주소들인지 검토된다.

9.3 SCTP 주소처리 함수

sctp_getpaddrs 함수

getpeername 함수는 멀티홈잉이 가능한 수송계층 프로토콜의 개념에는 맞지 않게 디자인되었다. SCTP에 사용하면 getpeername 함수는 단지 우선(primary) IP 주소만 반환한다. 모든 IP 주소들을 요구할 때 sctp_getpaddrs 함수는 상대방 단말의 모든 주소를 검색하기 위해 응용에서 메커니즘을 제공한다.

```
#include <netinet/sctp.h>

int sctp_getpaddrs( int sockfd, sctp_assoc_t id, struct sockaddr **addrs );

                    Returns: the number of peer addresses stored in addrs, -1 on error
```

*sockfd*는 소켓 디스크립터로써 socket 함수에 의해 반환된다. *id*는 일대다(one-to-many) 스타일 소켓에서 association 식별자이다. 일대일(one-to-one) 스타일 소켓에서는 *id* 필드는 무시된다. *addrs*는 IP 주소들의 포인터이고 내부적으로 할당되며 목록화된 IP 주소들로 sctp_getpaddrs 함수에 채워질 것이다. 종료할 때 sctp_getpaddrs에 의해 할당된 리소스를 해제하기 위해 sctp_freepaddrs를 사용해야만 한다.

sctp_freepaddrs 함수

sctp_freepaddrs 함수는 sctp_getpaddrs 함수에 의해 할당된 리소스를 해제한다.

```
#include <netinet/sctp.h>

void sctp_freepaddrs( sturct sockaddr *addrs );
```

addrs는 sctp_getpaddrs에 반환된 IP 주소들 배열의 포인터이다.

sctp_getladdrs 함수

sctp_getladdrs 함수는 association의 local IP 주소를 검색하는데 사용할 수 있다. 이 함수는 local 단말이 사용하고 있는 local IP 주소들을 정확히 알고 싶을 때 종종 필요하고 여기서 검색되는 local IP 주소들은 시스템 주소의 적당한 부분이 될 수도 있다.

```
#include <netinet/sctp.h>

int sctp_getladdrs( int sockfd, sctp_assoc_t id, struct sockaddr **addrs );

                Returns: the number of local addresses stored in addrs, -1 on error
```

*sockfd*는 소켓 디스크립터로써 socket 함수에 의해 반환된다. *id*는 일대다(one-to-many) 스타일 소켓에서 association 식별자이다. 일대일(one-to-one) 스타일 소켓에서는 *id* 필드는 무시된다. *addrs*는 IP 주소들의 포인터이고 내부적으로 할당되며 목록화된 IP 주소들로 sctp_getladdrs 함수에 채워질 것이다. 이 반환값의 구조체는 그림 9.4에서 상세히 보여준다. 종료할 때 sctp_getladdrs에 의해 할당된 리소스를 해제하기 위해 sctp_freeladdrs를 사용해야만 한다.

sctp_freeladdrs 함수

sctp_freeladdrs 함수는 sctp_getladdrs 함수에 의해 할당된 리소스를 해제한다.

```
#include <netinet/sctp.h>

void sctp_freeladdrs( struct sockaddr *addrs);
```

*addrs*는 sctp_getladdrs에 반환된 IP 주소들 배열의 포인터이다.

9.4 SCTP_SENDMSG 함수

응용은 sendmsg 함수를 14장에서 설명되어 있는 보조적인 데이터와 함께 사용함으로써 SCTP의 다양한 특성을 제어할 수 있다. 그러나 보조적인 데이타 사용이 불편할 수도 있으므로 많은 SCTP 구현에서는 시스템 콜로 구현된 보조의 라이브러리 콜을 제공하고 이 라이브러리 콜은 응용에서 SCTP 특성의 사용을 편리하게 한다. 그 형식은 다음을 따른다.

```
ssize_t sctp_sendmsg( int sockfd, const void *msg, size msgsz,
                      const struct sockaddr *to, socklen_t tolen,
                      uint32_t ppid,
                      uint32_t flags, uint16_t stream,
                      uint32_t timetolive, uint32_t context );
```

Returns: the number of bytes written, -1 on error

sctp_sendmsg 사용자는 더 많은 매개변수의 사용으로 전송 방법을 매우 간단히 한다. *sockfd* 필드는 socket 시스템 콜로부터 반환된 소켓 디스크립터를 가진다. *msg* 필드는 상대방 단말로 전송할 *msgsz* 바이트의 버퍼를 가리킨다. *tolen* 필드는 상대방 단말 주소의 길이를 가진다. *ppid* 필드는 데이터 청크와 통과될 pay-load protocol 식별자를 저장한다. *flags* 필드는 SCTP의 옵션을 식별하기 위해 SCTP 스택에 전달한다.

stream 필드는 SCTP에서의 스트림 수를 명시한다. *timetolive* 필드는 메시지의 lifetime을 millisecond 단위로 명시하고 *timetoleve* 필드가 0이라면 무한한 lifetime을 의미한다. 필요하다면 사용자 context는 *context* 필드에 명시된다. 사용자 context는 local 응용 context와 함께 메시지 통지를 통해 수신된 실패 메시지 전송을 associate 한다. 예를 들어, 스트림 수는 1개, 전송 플래그를 MSG_PR_SCTP_TTL로, lifetime은 1000 millisecond로, payload protocol 식별자는 24로, context는 52로 설정하여 메시지를 전송한다면 사용자는 아래 형식으로 호출한다.

```
ret = sctp_sendmsg( sockfd, data, datasz, &dest, sizeof(dest),
                    24, MSG_PR_SCTP_TTL, 1, 1000, 52 );
```

이 접근 방식은 필요한 보조 데이터를 할당하고 msghdr 구조체에 알맞은 구조를 설정하는 것보다 훨씬 쉽다. 구현을 할 때 sctp_sendmsg에서 sendmsg 함수 호출로 대체한다면 sendmsg 함수의 *flags* 필드는 0으로 설정한다.

9.5 SCTP_RECVMSG 함수

sctp_sendmsg와 유사하게 sctp_recvmsg 함수는 SCTP 특성을 사용하는데 있어 친근한 인터페이스를 제공한다. 이 함수의 사용은 상대방의 주소뿐만 아니라 *msg_flags* 필드를 얻게 한다.

msg_flags 필드는 recvmsg 함수에서 일반적으로 동반되는 필드로써 그 값은 MSG_NOTIFICATION, MSG_EOR 등이 있다. 또한 이 sctp_recvmsg 함수는 메시지 버퍼로부터 읽혀진 메시지와 함께하는 sctp_sndrcvinfo 구조체를 얻을 수 있다. 만약 응용이 sctp_sndrcvinfo 정보를 수신 받기 원한다면 sctp_data_io_event는 SCTP_EVENTS 소켓 옵션과 함께 설정되어야만 한다. sctp_recvmsg 함수는 아래와 같은 형식을 가진다.

```
ssize_t sctp_recvmsg( int sockfd, void *msg, size_t msgsz,
                      struct sockaddr *from, socklen_t *fromlen,
                      struct sctp_sndrcvinfo *sinfo, int *msg_flags );

                                 Returns: the number of bytes read, -1 on error
```

이 함수 호출로 반환된 *msg*는 *msgsz* 바이트의 데이터로 채워진다. 메시지 전송 주소는 *from* 필드에 저장되고 주소의 크기는 *fromlen* 매개변수에 저장된다. *msg_flags* 매개변수는 어떤 메시지 플래그들이라도 포함한다. sctp_data_io_event 통지가 사용 가능하도록 설정되었다면 sctp_sndrcvinfo 구조체는 메시지에 관한 상세한 정보로 채워질 것이다. 구현을 할 때 sctp_recvmsg에서 recvmsg 함수 호출로 대체한다면 sendmsg call의 *flags* 필드는 0으로 설정한다.

9.6 SCTP_OPT_INFO 함수

sctp_opt_info 함수는 SCTP에서 getsockopt 함수를 사용할 수 없는 구현을 위해 제공된다. SCTP_STATUS와 같은 association 식별자를 통해 in-out 변수를 필요로 하는 몇몇의 SCTP 소켓 옵션 때문에 getsockopt 함수는 사용할 수 없다. getsockopt 함수에서 in-out 변수를 제공할 수 없는 시스템에서는 sctp_opt_info 함수를 사용해야 한다. 소켓 옵션 콜에서 in-out 변수를 지원하는 FreeBSD 같은 시스템에서는 sctp_opt_info 함수는 적절한 getsockopt 함수로부터 매개변수를 재포장하는 라이브러리 콜이다. 이식성을 위해 응용은 in-out 변수를 요구하는 모든 옵션에서는 sctp_opt_info 함수를 사용해야만 한다.

이 함수는 다음과 같은 형식을 가진다.

```
int sctp_opt_info( int sockfd, sctp_assoc_t assoc_id, int opt,
                   void *arg, socklen_t *siz );
```

Returns: 0 for success, -1 on error

*sockfd*는 소켓 옵션에 영향을 받는 소켓 디스크립터이다. *assoc_id*는 옵션을 수행하는 association의 식별자이다. *opt*는 SCTP를 위한 소켓 옵션이다. *arg*는 소켓 옵션 매개변수이고 *siz*는 매개변수의 크기를 가지는 socklen_t 형태의 포인터이다.

9.7 SCTP_PEELOFF 함수

지난번에 언급한 바와 같이 일대다(one-to-many) 소켓의 association에서 단일의 일대일(one-to-one) 스타일의 소켓의 추출이 가능하다. 이 의미는 추가적인 매개변수를 포함한 accept 함수 호출과 유사하다. *sockfd*는 일대다(one-to-many) 소켓의 소켓 디스크립터이고 *id*는 추출되는 association 식별자이다. 이 호출이 정상적으로 수행되면 새로운 소켓 디스크립터를 반환한다. 이 새로운 소켓 디스크립터는 요구된 association에서의 일대일(one-to-one) 스타일을 가질 것이다. 이 함수는 다음과 같은 형식을 가진다.

```
int sctp_peeloff( int sockfd, sctp_assoc_t id );
```

Returns: a new socket descriptor on success, -1 on error

9.8 SHUTDOWN 함수

6절에서 언급한 shutdown 함수는 일대일(one-to-one) 스타일 인터페이스 기반의 SCTP 단말에서 사용할 수 있다. SCTP의 설계는 half-closed 상태를 제공하지 않기 때문에 shutdown 함수는 SCTP 단말에서 TCP 단말과는 다르게 동작한다. SCTP 단말이 중단 절차를 시작하면 양 단말에서는 큐에 있는 어떤 데이터라도 전송을 마쳐야 하고 association을 종료해야 한다. Active open으로 시작된 단말은 새로운 상대방과 연결을 하기 위해 close 대신에 shutdown 으로 사용하길 원한다. TCP와는 달리 새로운 소켓을 여는 close는 요구하지 않는다. SCTP는 shutdown 을 단말에게 허가하고 후에 shutdown을 끝내고, 단말은 그 사용한 소켓으로 새로운 상대방과 연결하여 재사용 할 수 있다. 단말이 SCTP 중단 절차가 끝날 때까지 기다리지 못한다면 새로운 연결은 실패할 것이다. 그림 9.5는 이 시나리오에서 전형적인 함수 호출을 보여준다.

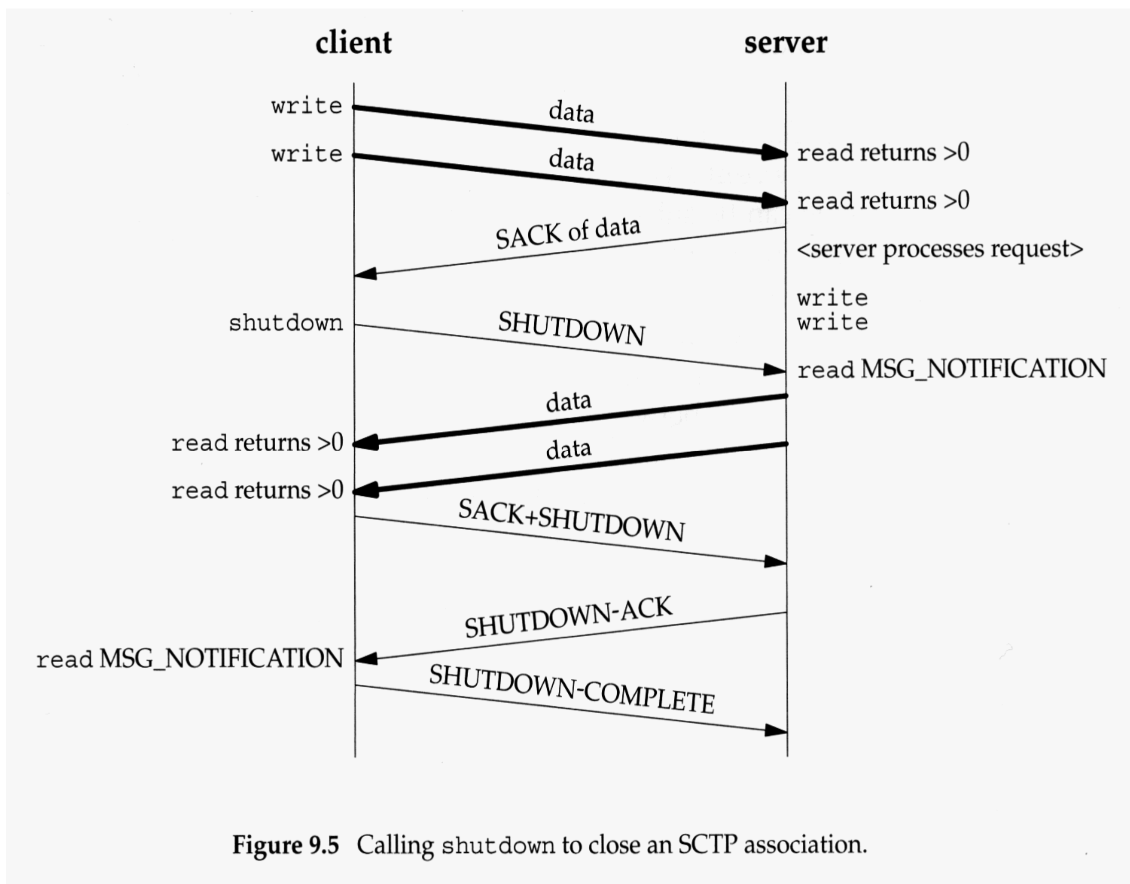


Figure 9.5 Calling shutdown to close an SCTP association.

그림 9.5에서는 MSG_NOTIFICATION 이벤트를 수신하는 것을 보여준다. 이 이벤트를 수신하는 것을 동의하지 않는다면 읽기의 길이는 0으로 반환된다. TCP에서 shutdown 함수의 결과는 6절에 기술되어 있다. shutdown 함수의 *howto* 필드는 SCTP에서 아래와 같은 의미를 가진다.

SHUT_RD 6.6절에서 언급한 TCP에서와 같은 동작을 수행한다. SCTP 프로토콜은 이 동작을 수행하지 않는다.

SHUT_WR 향후 전송 동작을 중단하고 association을 종료할 SCTP shutdown 절차를 시작한다. 이 옵션은 half-closed 상태를 제공하지 않지만 SCTP SHUTDOWN 메시지를 수신하기 전에 상대방이 전송한 큐에 있는 데이터는 local 단말이 읽을 수 있도록 허용한다.

SHUT_RDWR 모든 읽기와 쓰기 동작을 중단하고 SCTP shutdown 절차를 시작한다. local 단말에 전송된 큐에 있는 어떤 데이터라도 인정하고 나서 조용히 버린다.

9.9 NOTIFICATIONS

SCTP는 응용 프로그래머에게 유용하고 다양한 통지를 제공한다. SCTP 사용자는 이 통지들을 통해 association(s)의 상태를 추적할 수 있다. 통지들은 네트워크 상태 변화, association 시작, 원격 동작 오류, 전달할 수 없는 메시지를 포함한 수송계층 이벤트를 알린다. 일대일(one-to-one)과 일대다(one-to-many) 스타일에서 sctp_data_io_event를 제외한 모든 이벤트는 기본적으로 사용하지 않도록 되어 있다.

SCTP_EVENTS 소켓 옵션에서는 8개의 이벤트를 사용할 수 있다. 그 중의 7개 이벤트는 통지라 불리는 추가적인 데이터를 발생시키고 일반적인 소켓 디스크립터를 통해 수신 받을 수 있을 것이다. 그 통지는 그것들이 일어나도록 생성되는 이벤트로써 소켓 디스크립터에 데이터와 함께 즉시 추가된다. 소켓으로부터 통지 subscriptions을 읽었을 때 사용자 데이터와 통지들은 소켓 버퍼에 상호 배치되어 있을 것이다. 상대방 데이터와 통지를 구별하기 위해 recvmsg 함수나 sctp_recvmsg 함수를 사용한다. 반환된 데이터가 이벤트 통지이면, 이 두 함수의 *msg_flags* 필드는 MSG_NOTIFICATION 플래그가 포함되어 있을 것이다. 이 플래그는 응용에게 해당 메시지는 상대방으로부터의 데이터가 아니고 내부 SCTP 스택으로부터의 통지라는 것을 알려준다.

각 통지의 형식은 tag-length-value 형태를 가지며 메시지의 처음 8바이트는 도착한 통지의 형식이 무엇인가와 그것의 전체 길이를 식별한다. sctp_data_io_event 이벤트를 사용 가능하게 하는 것은 모든 읽혀진 사용자 데이터 상에서 sctp_sndrcvinfo 구조체를 받도록 하게 한다. 이 옵션은 두 가지 스타일의 인터페이스에서 기본적으로 사용 가능하다. 이 정보는 일반적으로 보조적인 데이터와 함께 recvmsg 콜을 사용하여 받을 수 있다. 응용은 sctp_recvmsg 콜을 사용할 수도 있고 sctp_sndrcvinfo 구조체는 이 정보와 함께 포인터로 채워질 것이다.

통지는 다음과 같은 형식을 가진다.

```
struct sctp_tlv {
  u_int16_t      sn_type;
  u_int16_t      sn_flags;
  u_int32_t      sn_length;
};

/* notification event */
union sctp_notification {
  struct sctp_tlv sn_header;
  struct sctp_assoc_change sn_assoc_change;
  struct sctp_paddr_change sn_paddr_change;
  struct sctp_remote_error sn_remote_error;
  struct sctp_send_failed sn_send_failed;
  struct sctp_shutdown_event sn_shutdown_event;
  struct sctp_adaption_event sn_adaption_event;
  struct sctp_pdapi_event sn_pdapi_event;
};
```

sn_header 필드는 형식 값을 해석하거나 보내진 실제 메시지를 해독하는데 사용한다. 그림 9.6은 *sn_header.sn_type* 필드의 명시된 값과 SCTP_EVENTS 소켓 옵션에서 대응되는 관련 필드들을 나타낸다.

sn_type	Subscription field
SCTP_ASSOC_CHANGE	sctp_association_event
SCTP_PEER_ADDR_CHANGE	sctp_address_event
SCTP_REMOTE_ERROR	sctp_peer_error_event
SCTP_SEND_FAILED	sctp_send_failure_event
SCTP_SHUTDOWN_EVENT	sctp_shutdown_event
SCTP_ADAPTION_INDOCATION	sctp_adaption_layer_event
SCTP_PARTIAL_DELIVERY_EVENT	sctp_partial_delivery_event

Figure 9.6 *sn_type* and event subscription field.

SCTP_ASSOC_CHANGE

이 통지는 새로운 association이 시작되거나 존재하는 association이 종료됨으로써 발생하는 변화를 응용에게 알린다. 그 정보는 다음과 같이 정의된 이벤트로 제공한다.

```
struct sctp_assoc_change {
  u_int16_t sac_type;
  u_int16_t sac_flags;
  u_int32_t sac_length;
  u_int16_t sac_state;
  u_int16_t sac_error;
  u_int16_t sac_outbound_streams;
  u_int16_t sac_inbound_streams;
  sctp_assoc_t sac_assoc_id;
  uint8_t sac_info[];
};
```

*sac_state*는 association에서 발생한 이벤트의 형식을 기술하고 다음 값 중 하나를 가질 것이다.

SCTP_COMM_UP	이 상태는 새로운 association이 막 시작되었다는 것을 나타낸다. inbound와 outbound 스트림 필드는 각 방향에 얼마나 많은 스트림이 사용 가능한지를 나타낸다. Association 식별자는 유일한 값으로 채워지며 이 association에 관련된 내부 SCTP 스택과 통신하는데 사용할 수 있다.
SCTP_COMM_LOST	이 상태는 association 식별자에 의해 명시된 association이 도달할 수 없는 한계에 도달하는 경우나(SCTP 단말에 여러 번 타임아웃이 발생하고 그것이 한계에 도달하면 상대방에게 더 이상 도달할 수 없다고 판단되는 경우) 상대방이 association의 종료를 실패할 경우의 종료를 나타낸다. 사용자가 명시한 정보는 *sac_info* 필드에 기록될 것이다.
SCTP_RESTART	이 상태는 상대방이 다시 시작하는 것을 나타낸다. 이 통지는 상대방이 중단되고 다시 시작될 때 주로 발생된다. 응용은 다시 시작 하는 동안 각 방향의 스트림 수가 변할 수 있으므로 검증해야만 한다.

SCTP_SHUTDOWN_COMP 이 상태는 내부 단말에서 shutdown 콜이나 MSG_EOF 플래그를 포함한 sendmsg에 의해 시작된 shutdown이 완전히 종료되었을 때를 나타낸다. 일대일(one-to-one) 스타일에서는 이 통지를 받은 후에 소켓 디스크립터는 다른 상대방과 다시 연결을 맺는데 사용할 수 있다.

SCTP_CANT_STR_ASSOC 이 상태는 상대방이 INIT 메시지와 같은 association 설정 시도에 대한 응답이 없을 때를 나타낸다.

sac_error 필드는 association 변화에 따라 발생할 수 있는 SCTP 프로토콜 오류 원인 코드를 담고 있다. *sac_outbound_streams*와 *sac_inbound_streams* 필드는 association 상에서 각 방향으로 얼마나 많은 스트림을 사용하는지 협상한 정보를 응용에게 전달한다. *sac_assoc_id*는 association을 위한 유일한 식별자를 담고 있고 소켓 옵션과 향후 통지에서 association을 식별하는데 사용할 수 있다. *sac_info*는 사용자가 이용 가능한 다른 정보를 담고 있다. 예를 들어 association이 사용자가 정의한 오류로 중단되면 오류는 이 필드에 저장될 것이다.

SCTP_PEER_ADDR_CHANGE

이 통지는 상대방의 주소들 중 하나의 상태 변화가 발생되면 알려준다. 이 변화는 전송 했을 때 도착지에서 응답이 없는 실패나 실패 상태에서 복구된 도착지 복구를 의미한다. 주소 변화를 수행하는 구조체는 다음과 같다.

```
struct sctp_paddr_change {
  u_int16_t spc_type;
  u_int16_t spc_flags;
  u_int32_t spc_length;
  struct sockaddr_storage spc_aaddr;
  u_int32_t spc_state;
  u_int32_t spc_error;
  sctp_assoc_t spc_assoc_id;
};
```

spc_addrs 필드는 이 이벤트에 영향을 받는 상대방의 주소를 담고 있다. *spc_state* 필드는 그림 9.7에 있는 값 중의 하나를 담고 있다.

spc_state	Description
SCTP_ADDR_ADDED	Address is now added to the association
SCTP_ADDR_AVAILABLE	Address is now reachable
SCTP_ADDR_CONFIRMED	Address has now been confirmed and is valid
SCTP_ADDR_MADE_PRIM	Address has now been made the primary destination
SCTP_ADDR_REMOVED	Address is no longer part of the association
SCTP_ADDR_UNREACHABLE	Address can no longer be reached

Figure 9.7 SCTP peer address state notifications.

하나의 주소가 SCTP_ADDR_UNREACHABLE으로 선언되면 재설정한 경로인 대체주소(alternate address)로 데이터를 전송한다. 또한 SCTP_ADDR_ADDED나 SCTP_ADDR_REMOVED와 같은 몇 가지 상태는 동적 주소 옵션을 지원하는 SCTP 구현상에서만 사용 가능하다.

spc_error 필드는 그 이벤트에 관한 더 많은 정보를 제공하기 위해 통지 오류 코드를 포함하고, *spc_assoc_id*는 association 식별자를 담고 있다.

SCTP_REMOTE_ERROR

원격 상대방은 내부 단말에게 부가적인 오류 메시지를 전송할 수도 있다. 이 메시지들은 association에서 다양한 오류 조건을 나타낼 수 있다. 전체 오류 청크는 이 통지가 사용 가능할 때 유선 환경에서 응용에게 전달될 수 있다. 그 메시지의 형식은 다음과 같다.

```
struct sctp_remote_error {
  u_int16_t sre_type;
  u_int16_t sre_flags;
  u_int32_t sre_length;
  u_int16_t sre_error;
  sctp_assoc_t sre_assoc_id;
  u_int8_t sre_data[];
};
```

*sre_error*는 SCTP 프로토콜 오류 원인 코드들 중 하나를 담고 있고, *sre_assoc_id*는 association 식별자를 포함한다. 그리고 *src_data*는 유선 환경에서의 전체 오류를 담고 있다.

SCTP_SHUTDOWN_EVENT

이 통지는 상대방이 우리의 내부 단말에게 SHUTDOWN 청크를 전송할 때 응용에게 전달된다. 이 통지는 소켓상에서 더 이상 새로운 데이터가 받아들여지지 않는다는 것을 응용에게 알려준다. 모든 현재 큐의 데이터는 모두 전송될 것이고 그 전송이 종료되면 association은 중단될 것이다. 이 통지의 형식은 아래와 같다.

```
struct sctp_shutdown_event {
  uint16_t sse_type;
  uint16_t sse_flags;
  uint32_t sse_length;
  sctp_assoc_t sse_assoc_id;
};
```

SCTP_ADAPTION_INDICATION

어떤 구현들은 adaption layer indication 매개변수를 제공한다. 이 매개변수는 각각의 상대방에게 어떤 형태의 application adaption이 수행되고 있는지를 알리기 위해 INIT과 INIT-ACK에서 교환된다. 통지는 다음과 같은 형식을 가진다.

```
struct sctp_adaption_event {
  u_int16_t sai_type;
  u_int16_t sai_flags;
  u_int32_t sai_length;
  u_int32_t sai_adaption_ind;
  sctp_assoc_t sai_assoc_id;
};
```

sai_assoc_id 필드는 adaption layer notification의 association를 식별한다. *sai_adaption_ind* 필드는 상대방이 INIT과 INIT-ACK 메시지에 실어 우리의 내부 호스트에게 전달하는 32비트 정수이다. Outgoing adaption layer는 SCTP_ADAPTION_LAYER 소켓 옵션으로 설정될 수 있다.

10. SCTP CLIENT/SERVER 예제

완전한 일대다(one-to-many) SCTP 서버/클라이언트 예제를 작성하기 위해 9장에서의 기본 함수들을 사용할 것이다. 간단한 예제는 5장에 있는 에코 서버와 유사하고 다음의 단계를 수행한다.

1. 클라이언트는 표준 입력으로 하나의 텍스트 라인을 읽어 서버에게 전송한다. 그 라인은 [#] text 형식을 따르고 꺾음 괄호 안에 숫자는 텍스트 메시지가 전송되어야만 하는 SCTP 스트림 번호이다.
2. 서버는 네트워크로부터 그 텍스트 메시지를 수신하고, 도착한 메시지의 스트림 번호를 하나 증가시키다. 그리고 이 새로운 스트림 번호로 클라이언트에게 텍스트 메시지를 다시 전송한다.
3. 클라이언트는 에코된 라인을 읽고 표준 출력으로 스트림 번호, 스트림 일련 번호, 텍스트 스트림을 출력한다.

그림 10.1은 입력과 출력을 위해 사용되는 함수들과 간단한 서버/클라이언트를 보여주고 있다.

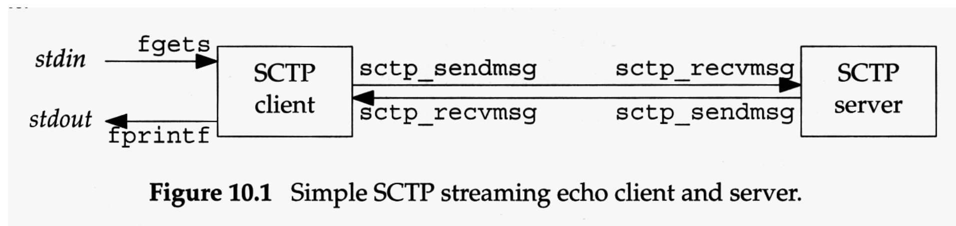


Figure 10.1 Simple SCTP streaming echo client and server.

전체 association이 full-duplex이지만 사용되는 두 개의 단방향 스트림을 묘사하기 위해 서버와 클라이언트 사이에 두 개의 화살표로 보여준다. fgets와 fputs 함수는 표준 I/O 라이브러리이다. 3절에서 정의한 writen과 readline 함수는 불필요하여 사용하지 않는다. 대신에 9절에서 정의한 sctp_sendmsg와 sctp_recvmsg 함수는 각각 사용한다.

이 예제에서는 일대다(one-to-many) 스타일 서버를 사용한다. 5장의 예제들은 socket 함수에서 3번째 매개변수를 IPPROTO_TCP에서 IPPROTO_SCTP로 변경하면 SCTP상에서 동작하도록 할 수 있다.

10.1 SCTP ONE-TO-MANY STYLE STREAMING ECHO SERVER

SCTP 서버/클라이언트는 그림 10.2에 도식화된 함수 흐름을 따른다. 10.2절에서는 순차접속 (iterative) 서버 프로그램을 보여준다.

sctp/sctpserv01.c

```
 1 #include   "lnp.h"

 2 int
 3 main( int argc, char **argv )
 4 {
 5    int    sock_fd, msg_flags;
 6    char   readbuf[BUFFSIZE];
 7    struct sockaddr_in servaddr, cliaddr;
 8    struct sctp_sndrcvinfo sri;
 9    struct sctp_event_subscribe evnts;
10    int    stream_increment = 1;
11    socklen_t len;
12    size_t rd_sz;
13    if( argc == 2 )
14       stream_increment = atoi( argv[1] );
15    sock_fd = Socket( AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP );
16    bzero( &servaddr, sizeof(servaddr) );
17    servaddr.sin_family = AF_INET;
18    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
19    servaddr.sin_port = htons(SERV_PORT);

20    Bind( sock_fd, (SA *) &servaddr, sizeof(servaddr) );

21    bzero( &evnts, sizeof(evnts) );
22    evnts.sctp_data_io_event = 1;
23          Setsockopt(  sock_fd,  IPPROTO_SCTP,  SCTP_EVENTS,  &evnts,
sizeof(evnts) );

24    Listen( sock_fd, LISTENQ );
25    for( ;; ) {
26       len = sizeof(struct sokaddr_in);
27       rd_sz = Sctp_recvmsg( sock_fd, readbuf, sizeof(readbuf),
28                             (SA *) &cliaddr, &len, &sri, &msg_flags );
29       if( stream_increment ) {
30          sri.sinfo_stream++;
31          if( sri.sinfo_stream >=
32             sctp_get_no_strms( sock_fd, (SA *) &cliaddr, len ) )
33             sri.sinfo_stream = 0;
34       }
35       Sctp_sendmsg( sock_fd, readbuf, rd_sz,
36                     (SA *) &cliaddr, len,
37                     sri.sinfo_ppid,
38                     sri.sinfo_flags, sri.sinfo_stream, 0, 0 );
39    }
40 }
```

Figure 10.2 SCTP streaming echo server

Set stream increment option

13-14 기본적으로 서버는 메시지를 수신 받을 때의 스트림 번호보다 하나 높은 스트림 번호를 사용하여 반응한다. 정수 매개변수가 명령어 라인으로 입력되면 서버는 stream_increment의 값으로 그 매개변수를 해석하고 들어오는 메시지의 스트림 번호를 증가시킬지 시키지 않을지를 결정한다. 뒤에서 head-of-line blocking의 토론에서 이 옵션을 사용할 것이다.

Create an SCTP socket

15 SCTP 일대다(one-to-many) 스타일 소켓이 생성되었다.

Bind an address

16-20 인터넷 소켓 주소 구조체는 와일드카드 주소(INADDR_ANY)와 서버의 well-known 포트인 SERV_PORT로 채워진다. 와일드카드 주소의 바인딩은 SCTP 단말이 설정한 association의 사용 가능한 모든 로컬 주소들을 사용할 것이라고 시스템에게 알려준다. 멀티홈잉이 가능한 호스트에서 바인딩은 원격 단말은 association들을 만들 수 있고 로컬 호스트의 아무 경로 가능한 주소들에게 패킷을 전송할 수 있다는 것을 의미한다. 이 서버는 5절에 있는 이전 예제에서 먼저 고려된 부분이 똑같이 고려된다.

Set up for notifications of interest

21-23 서버는 일대다(one-to-many) SCTP 소켓을 위한 통지 subscription을 변경한다. 서버는 단지 sctp_data_io_event만 동의하고 서버가 sctp_sndrcvinfo 구조체를 보는 것을 허가할 것이다. 이 구조체로부터 서버는 도착된 메시지에서 스트림 번호를 결정할 수 있다.

Enable incoming associations

24 서버는 listen 콜을 이용해 들어오는 association들을 가능하게 한다. 그리고 나서 제어는 메인 프로세싱 루프로 들어간다.

Wait for message

26-28 서버는 클라이언트 소켓 주소 구조체의 크기를 초기화하고 원격 상대방으로부터 메시지를 기다리는 동안 block한다.

Increment stream number if desired

29-34 메시지가 도착했을 때, 서버는 스트림 번호를 증가시켜야 하는지 stream_increment 플래그를 검사한다. 플래그가 설정되어 있으면(명령어 라인을 통한 매개변수가 없는 경우), 서버는 메시지의 스트림 번호를 증가시킨다. 번호가 최대 스트림과 크거나 같으면 sctp_get_no_strms 내부 함수를 호출하고, 서버는 스트림을 0으로 재설정한다. sctp_get_no_strms 함수는 보여지지 않는다. 교섭된 스트림 번호를 찾기 위해 SCTP_STATUS SCTP 소켓 옵션을 사용한다.

Send back response

35-38 서버는 sri 구조체를 이용하여 payload 프로토콜 ID, 플래그, 변경된 스트림 번호를 포함한 메시지를 전송한다.

이 서버는 association 통지를 원하지 않으므로 소켓 버퍼에서 메시지를 통해 올라온 모든 이벤트를 사용 가능하지 않도록 한다. 서버는 sctp_sndrcvinfo 구조체에서의 정보와 상대방 association이 정한 cliaddr에 반환된 주소와 반환된 에코를 신뢰한다.

이 프로그램은 사용자가 외부적 중단 신호를 줄 때까지 영구히 동작한다.

10.2 SCTP ONE-TO-MANY STYLE STREAMING ECHO CLIENT

그림 10.3은 SCTP 클라이언트 메인 함수를 보여준다.

Validate arguments and create a socket

9-15 클라이언트는 이것을 통해 매개변수를 확인한다. 먼저 클라이언트는 호스트에게 메시지 전송을 제공하는지 검증한다. 그리고 "echo to all" 옵션이 사용 가능한지를 확인한다. 마지막으로 클라이언트는 SCTP 일대다(one-to-many) 스타일 소켓을 생성한다.

Set up server address

16-20 클라이언트는 inet_pton 함수를 이용하여 명령어 라인으로 입력된 서버 주소를 변환한다. 그 주소는 서버의 well-known 포트 번호와 결합하여 요청을 위한 목적지 주소로 사용된다.

sctp/sctpclient01.c

```
 1 #include   "lnp.h"

 2 int
 3 main( int argc, char **argv )
 4 {
 5    int    sock_fd;
 6    struct sockaddr_in servaddr;
 7    struct sctp_event_subscribe evnts;
 8    int   echo_to_all = 0;
 9    if( argc < 2 )
10          err_quit( "Missing host argument - use '%s host [echo]'\n",
argv[0] );
11    if( argc > 2 ) {
12        printf( "Echoing messages to all streams\n" );
13        echo_to_all = 1;
14    }
15    sock_fd = Socket( AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP );
16    bzero( &servaddr, sizeof(servaddr) );
17    servaddr.sin_family = AF_INET;
18    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
19    servaddr.sin_port = htons(SERV_PORT);
20    Inet_pton( AF_INET, argv[1], &servaddr.sin_addr );

21    bzero( &evnts, sizeof(evnts) );
22    evnts.sctp_data_io_event = 1;
23          Setsockopt(  sock_fd,  IPPROTO_SCTP,  SCTP_EVENTS,  &evnts,
sizeof(evnts) );
24    if( echo_to_all == 0 )
25                  sctpstr_cli(  stdin,  sock_fd,  (SA  *)  &servaddr,
sizeof(servaddr) );
26    else
27        sctpstr_cli_echoall( stdin, sock_fd, (SA *) &servaddr,
28                              sizeof(servaddr) );
29    Close(sock_fd);
30    return (0);
31 }
```

Figure 10.3 SCTP streaming echo client main()

Set up for notifications of interest

21-23 클라이언트는 일대다(one-to-many) SCTP 소켓에 의해 제공되는 통지 subscription을 명시적으로 설정한다. MSG_NOTIFICATION 이벤트가 아닌 것을 원한다. 그러므로 클라이언트는 서버에서 했던 것와 같이 MSG_NOTIFICATION 이벤트를 사용하지 않도록 하고 오직 sctp_sndrcvinfo 구조체의 수신을 가능하게 한다.

Call echo processing function

24-28 echo_to_all 플래그가 설정되지 않았다면 클라이언트는 sctpstr_cli 함수를 호출한다. echo_to_all 플래그가 설정되었다면 클라이언트는 sctpstr_cli_echoall 함수를 호출한다. SCTP 스트림 사용을 조사하는 것처럼 10.5절에서 이 함수는 이야기될 것이다.

Finish up

29-31 반환 과정으로 클라이언트는 SCTP 소켓을 종료하고, 소켓을 이용하는 SCTP association을 중단시킨다. 클라이언트는 메인에서 반환 코드로 프로그램이 성공적으로 동작했다는 0을 반환한다.

10.3 SCTP STREAMING ECHO CLIENT

그림 10.4는 SCTP 기본 클라이언트 처리 함수를 보여준다.

```
sctp/sctp_strcli.c

 1 #include    "lnp.h"

 2 void
 3 sctpstr_cli( FILE *fp, int sock_fd, struct sockaddr *to, socklen_t
tolen )
 4 {
 5     struct sockaddr_in peeraddr;
 6     struct sctp_sndrcvinfo sri;
 7     char    sendline[MAXLINE], recvline[MAXLINE];
 8     socklen_t len;
 9     int     out_sz, rd_sz;
10     int     msg_flags;

11     bzero( &sri, sizeof(sri) );
12     whlie( fgets(sendline, MAXLINE, fp) != NULL ) {
13         if( sendline[0] != '[' ] {
14                 printf( "Error, line must be of the form '[streamnum]
text'\n" );
15             continue;
16         }
17         sri.sinfo_stream  = strtol( &sendline[1], NULL, 0 );
18         out_sz = strlen(sendline);
19         Sctp_sendmsg( sock_fd, sendline, out_sz,
20                       to, tolen, 0, 0, sri.sinfo_stream, 0, 0 );

21         len = sizeof(peeraddr);
22         rd_sz = Sctp_recvmsg( sock_fd, recvline, sizeof(recvline),
23                                             (SA *) &peeraddr, &len, &sri,
&msg_flags );
24         printf( "From str:%d seq:%d (assoc:0x%x):",
25                             sri.sinfo_stream, sri.sinfo_ssn, (u_int)
sri.sinfo_assoc_id );
26         printf( "%.*s", rd_sz, recvline );
27     }
28 }
```

Figure 10.4 SCTP sctp_strcli function

Initialize the sri structure and enter loop

11-12 클라이언트는 sctp_sndrcvinfo 구조체인 sri를 초기화하면서 시작한다. 클라이언트는 blocking 함수인 fget을 통한 fp로부터 읽어 들이는 루프로 들어간다. 메인 프로그램은 이 함수에서 stdin을 전달하고, 사용자 입력은 사용자에 의해 종료 EOF 문자(Control-D)를 입력될 때까지 루프안에서 읽어 들이고 처리된다. 이 사용자 동작은 함수를 종료하고 반환시킨다.

Validate input

13-16 클라이언트는 사용자 입력이 [#] text 형식이 확실한지를 검사한다. 만약 형식이 유효하지 않다면 클라이언트는 오류 메시지를 출력하고 fget의 blocking 상태로 되돌아간다.

Translate stream number

17 클라이언트는 입력으로 사용자가 요구한 스트림를 sri 구조체의 sinfo_stream 필드로 변환한다.

Send message

18-20 주소의 길이와 실제 사용자 데이터의 크기를 알맞게 초기화한 후에 클라이언트는 sctp_sendmsg 함수를 사용하여 메시지를 전송한다.

Block while waiting for message

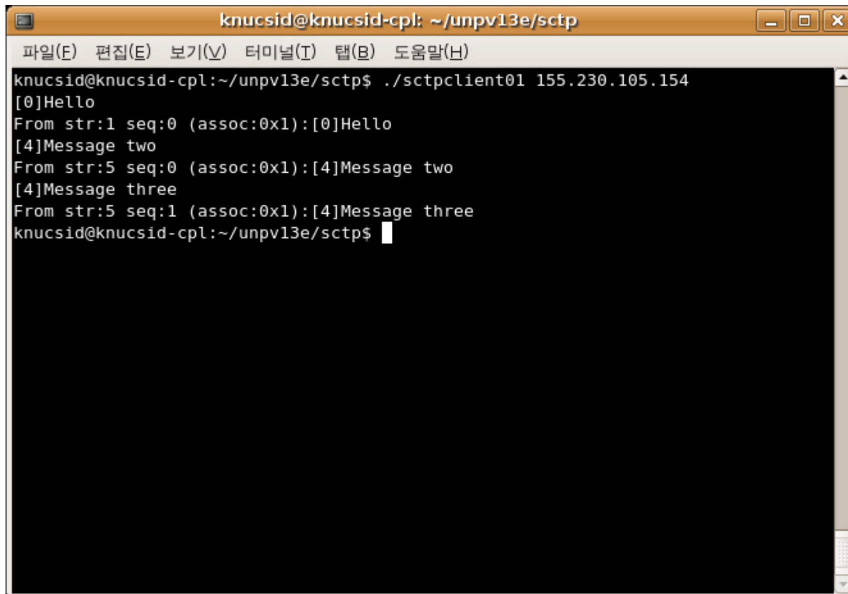
21-23 클라이언트는 서버로부터 에코 메시지가 도착할 때가지 block하고 기다린다.

Display returned message and loop

24-26 클라이언트는 반환된 에코 메시지에서 스트림 번호, 스트림 일련 번호, 텍스트 메시지를 보여준다. 이 메시지를 보여준 후 클라이언트는 사용자로부터 다른 요구를 입력 받기 위해 루프로 되돌아간다.

Running the Code

사용자는 SCTP 에코 서버를 FreeBSD 머신 상에서 매개변수 없이 시작한다. 클라이언트는 서버의 주소를 매개변수로 하여 시작된다.



클라이언트는 스트림 번호 0과 4로 메시지를 전송하는데 서버는 스트림 번호 1과 5로 메시지를 전송한다. 이 동작은 매개변수가 없을 경우 동작된다. 또한 스트림 일련 번호는 기대했던 것처럼 스트림 번호 5로 수신된 두번째 메시지에 증가되었다.

10.4 HOL(HEAD-OF-LINE) BLOCKING

간단한 서버는 많은 스트림 중 어느 곳으로든 텍스트 메시지를 전송하는 방법을 제공한다. SCTP의 스트림은 TCP에서처럼 바이트 스트림이 아니고 association의 범위 내에서 순서 있는 일련 메시지이다. 이것의 하위 순차 스트림들은 TCP에서의 head-of line blocking을 피하는데 사용된다.

Head-of-line blocking은 TCP 세그먼트가 손실되고 순서를 벗어나는 다음의 TCP 세그먼트가 도착할 때 발생한다. 다음의 세그먼트는 첫번째 TCP 세그먼트가 재전송되고 수신 측에 도착할 때까지 가지고 있는다. 다음 세그먼트의 전달 지연은 수신하는 응용이 전송하는 응용이 보낸 순서대로 데이터를 보는 것을 보장한다. 완전한 순서를 위한 지연은 꽤 유용하지만 부정적인 면을 가진다. 단일 TCP 연결상에서 독립적인 메시지를 전송한다고 가정하자. 예를 들어, 서버는 웹 브라우저에 보여주기 위해 세 개의 다른 그림을 전송할 것이다. 그 그림을 사용자 스크린 상에 병렬적으로 나타내기 위해, 서버는 첫번째 그림의 조각을 전송하고, 두번째 그림의 조각을 전송하고 마지막으로 세번째 그림의 조각을 전송한다. 서버는 세 개의 그림이 모두 브라우저에게 성공적으로 전송될 때까지 이 과정을 반복한다. 그러나 첫번째 그림의 조각을 가지는 TCP 패킷이 손실되면 무슨

일이 발생할까? 클라이언트는 받지 못한 조각이 재전송되고 성공적으로 도착할 때까지 모든 데이터를 가지고 있을 것이고 첫번째 그림의 데이터와 마찬가지로 두번째와 세번째 그림의 데이터는 지연될 것이다.

그림 10.5는 이 문제를 보여주고 있다.

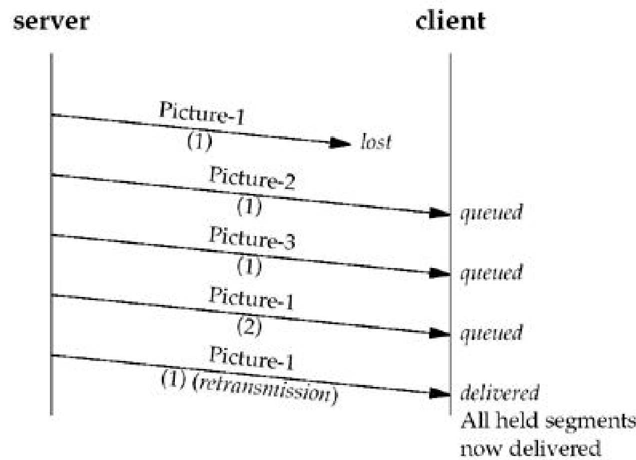


Figure 10.5 Sending three pictures over one TCP connection

비록 HTTP 클라이언트가 일반적으로 동작하는 것과 같이 그림 하나당 하나의 TCP 연결을 생성하는 것은 head-of-line blocking을 피할 수 있지만 각 연결은 RTT와 사용 가능한 대역폭을 알고 있어야 한다. 한 연결에서의 손실은 반드시 다른 연결의 속도를 낮추지는 않는다. 이것은 혼잡이 있는 네트워크에서의 전체 사용성은 낮게 한다.

이 blocking은 실제로 응용에서 발생시키려고 하는 것은 아니다. 이상적으로는, 순서대로 도착한 두 번째와 세 번째 그림의 조각들은 사용자에게 즉시 전달되는 반면에 단지 첫 번째 그림의 이후 조각들만 지연될 것이다.

Head-of-line blocking은 SCTP의 다중 스트림 특성으로 최소화할 수 있다. 그림 10.6에서 3개의 그림을 전송하고 있다. 이 때 서버는 두 번째와 세 번째 그림의 전달을 허가하지만 순차적 전달이 가능할 때까지 부분적으로 수신된 첫 번째 그림은 가지고 있도록 하는 곳에서만 단지 head-of-line blocking이 발생하도록 스트림들을 사용한다.

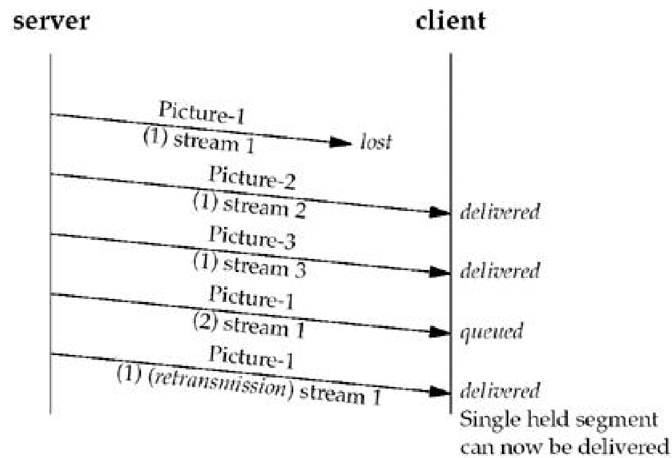


Figure 10.6 Sending three pictures over three SCTP streams

빠진 sctpstr_cli_echoall 함수를 포함하는 완전한 클라이언트 코드는 어떻게 SCTP가 head-of-line blocking을 최소화하는지를 증명하는데 사용할 것이다. 이 함수는 클라이언트가 더 이상 이전의 각각의 메시지에서 괄호 안의 스트림 번호를 기대하지 않는 것을 제외하고는 이전의 sctpstr_cli 함수와 유사하다. 대신에 이 함수는 모든 SERV_MAX_SCTP_STRM 스트림들로 사용자 메시지를 전송한다. 메시지를 전송한 후에 클라이언트는 서버로부터 도착 응답을 기다린다. 동작하는 코드에서 서버는 메시지가 수신된 같은 스트림으로 응답하기 위해 서버에게 추가적인 매개변수를 넘겨줄 수 있다. 이 방법은 사용자가 응답 전송과 그들의 도착 순서를 더 추적할 수 있다.

Initialize data structures and wait for input

13-15 이전처럼, 클라이언트는 sri 구조체는 송신과 수신을 하기 위한 스트림을 설정하기 위해 초기화한다. 그리고 클라이언트는 사용자 입력을 받아들이는 데이터 버퍼를 0으로 초기화한다. 그리고 나서 클라이언트는 메인 루프로 들어가고 blocking 된다.

Pre-process message

16-20 클라이언트는 메시지 크기를 설정하고 버퍼의 끝의 개행 문자를 삭제한다.

```
sctp/sctp_strcliecho.c

 1 #include   "lnp.h"

 2 #define SCTP_MAXLINE   800

 3 void
 4 sctpstr_cli_echoall( FILE *fp, int sock_fd, struct sockaddr *to,
 5                      socklen_t tolen )
 6 {
 7    struct sockaddr_in peeraddr;
 8    struct sctp_sndrcvinfo sri;
 9    char   sendline[SCTP_MAXLINE], recvline[SCTP_MAXLINE];
10    socklen_t len;
11    int    rd_sz, I, strsz;
12    int    msg_flags;

13    bzero( sendline, sizeof(sendline) );
14    bzero( &sri, sizeof(sri) );
15    while( fgets(sendline, SCTP_MAXLINE - 9, fp) != NULL ) {
16       strsz = strlen(sendline);
17       if( sendline[strsz - 1] == '\n' ) {
18          sendline[strsz - 1] = '\0';
19          strsz--;
20       }
21       for( i = 0; i < SERV_MAX_SCTP_STRM; i++ ) {
22          snprintf( sendline + strsz, sizeof(sendline) - strsz,
23                    ".msg. %d", i );
24          Sctp_sendmsg( sock_fd, sendline, sizeof(sendline),
25                        to, tolen, 0, 0, i, 0, 0 );
26       }
27       for( i = 0, i < SERV_MAX_SCTP_STRM; i++ ) {
28          len = sizeof(peeraddr);
29          rd_sz = Sctp_recvmsg( sock_fd, recvline, sizeof(recvline),
30                          (SA *) &peeraddr, &len, &sri, &msg_flags );
31          printf( "From str:%d se:%d (assoc: 0x%x): ",
32                  sri.sinfo_stream, sri.sinfo_ssn,
33                  (u_int) sri.sinfo_assoc_id );
34          printf( "%.*s\n", rd_sz, recvline );
35       }
36    }
37 }
```

Figure 10.7 sctp_strcliecho

Send message to each stream

21-26 클라이언트는 sctp_sendmsg 함수를 이용하여 메시지를 전송하는데 SCTP_MAXLINE 바이트의 전체 버퍼를 전송한다. 메시지를 전송하기 전에 도착하는 메시지의 순서를 식별하기 위하여 그 메시지에 ".msg." 문자열과 스트림 번호를 덧붙인다. 이 방법은 클라이언트가 실제 메시지를 전송할 때의 순서와 도착한 순서를 비교할 수 있다. 또한 클라이언트는 실제로 얼마나 많이 설정했는지 관계 없이 설정한 스트림 번호로 메시지를 전송한다. 상대방이 스트림 수를 아래쪽으로 협의한다면 하나 또는 그 이상의 전송은 실패할 수도 있다.

전송 또는 수신 윈도우가 너무 작다면 이 코드는 잠재적으로 실패를 가진다. 상대방의 수신 윈도우가 너무 작다면 클라이언트는 block될 수 있다. 클라이언트는 모든 전송이 완전히 끝날 때까지 어떤 정보도 읽지 못하기 때문에 서버는 이미 보낸 응답에 대해 읽기를 마치는 클라이언트를 기다리는 동안 잠재적으로 block될 수 있다. 그 같은 시나리오의 결과로 두 단말에서 deadlock이 발생할 수 있다. 이 코드는 확장성이 없는 대신에 간단한 방법으로 스트림과 head-of-line blocking을 설명한다.

Read back echoed messages and display

27-35 서버로부터 모든 대응 메시지를 읽고 전처럼 보여준다. 마지막 메시지를 읽은 후에 클라이언트는 사용자 입력을 위해 루프로 되돌아간다.

Running the Code

별개의 구성이 가능한 라우터와 두 개의 별개 FreeBSD 머신에서 서버와 클라이언트를 실행한다. 라우터는 지연시간과 손실을 조절할 수 있다. 처음 프로그램은 라우터에 의한 손실 없이 실행한다.

서버는 추가적인 매개변수 0고 함께 시작하고 서버는 강제로 응답 스트림 번호를 증가시킬 수 없다. 그 후, 각 스트림으로 메시지를 전송할 에코 서버의 주소와 추가적인 매개변수와 함께 클라이언트를 시작한다.

```
knucsid@knucsid-cpl: ~/unpv13e/sctp
파일(F) 편집(E) 보기(V) 터미널(T) 탭(B) 도움말(H)
knucsid@knucsid-cpl:~/unpv13e/sctp$ ./sctpclient01 192.168.62.20 echo
Echoing messages to all streams
Hello
From str:1 seq:0 (assoc:0x1):Hello.msg.0
From str:2 seq:0 (assoc:0x1):Hello.msg.1
From str:3 seq:0 (assoc:0x1):Hello.msg.2
From str:4 seq:0 (assoc:0x1):Hello.msg.3
From str:5 seq:0 (assoc:0x1):Hello.msg.4
From str:6 seq:0 (assoc:0x1):Hello.msg.5
From str:7 seq:0 (assoc:0x1):Hello.msg.6
From str:8 seq:0 (assoc:0x1):Hello.msg.7
From str:9 seq:0 (assoc:0x1):Hello.msg.8
From str:0 seq:0 (assoc:0x1):Hello.msg.9

knucsid@knucsid-cpl:~/unpv13e/sctp$
```

손실 없이, 클라이언트는 서버에게 보낸 순서로 대응 메시지가 도착하는 것을 볼 수 있다. 라우터의 매개변수를 모든 방향에서 모든 패킷의 10%가 손실되도록 변경시키고 다시 클라이언트를 시작한다.

```
knucsid@knucsid-cpl: ~/unpv13e/sctp
파일(F) 편집(E) 보기(V) 터미널(T) 탭(B) 도움말(H)
knucsid@knucsid-cpl:~/unpv13e/sctp$ ./sctpclient01 192.168.62.20 echo
Echoing messages to all streams
Hello
From str:2 seq:0 (assoc:0x1):Hello.msg.1
From str:3 seq:0 (assoc:0x1):Hello.msg.2
From str:4 seq:0 (assoc:0x1):Hello.msg.3
From str:1 seq:0 (assoc:0x1):Hello.msg.0
From str:6 seq:0 (assoc:0x1):Hello.msg.5
From str:7 seq:0 (assoc:0x1):Hello.msg.6
From str:5 seq:0 (assoc:0x1):Hello.msg.4
From str:8 seq:0 (assoc:0x1):Hello.msg.7
From str:9 seq:0 (assoc:0x1):Hello.msg.8
From str:0 seq:0 (assoc:0x1):Hello.msg.9

knucsid@knucsid-cpl:~/unpv13e/sctp$
```

스트림 안의 메시지들은 클라이언트가 각 스트림으로 두 개의 메시지를 전송함으로써 재배열을 위해 적절하게 잡혀 있는 것을 검증할 수 있다. 클라이언트는 각 메시지 중복을 식별하기 위해 그것의 메시지 번호를 더하도록 변경한다.

sctp/sctp_strcliecho2.c

```
21        for( i = 0; i < SERV_MAX_SCTP_STRM; i++ ) {
22           snprintf( sendline + strsz, sizeof(sendline) - strsz,
23                     ".msg.%d 1", i );
24           Sctp_sendmsg( sock_fd, sendline, sizeof(sendline),
25                         to, tolen, 0, 0, i, 0, 0 );
26           snprintf( sendline + strsz, sizeof(sendline) - strsz,
27                     ".msg.%d 2", i );
28           Sctp_sendmsg( sock_fd, sendline, sizeof(sendline),
29                         to, tolen, 0, 0, i, 0, 0 );
30        }
31        for( i = 0; i < SERV_MAX_SCTP_STRM * 2; i++) {
32           len = sizeof(peeraddr);
```

Add additional message number and send

22-25 클라이언트는 메시지 전송되는 것을 추적하기 위해 추가적인 메시지 번호 1을 더한다. 그리고 나서 클라이언트는 sctp_sendmsg 함수를 이용하여 메시지를 전송한다.

Change message number and send it again

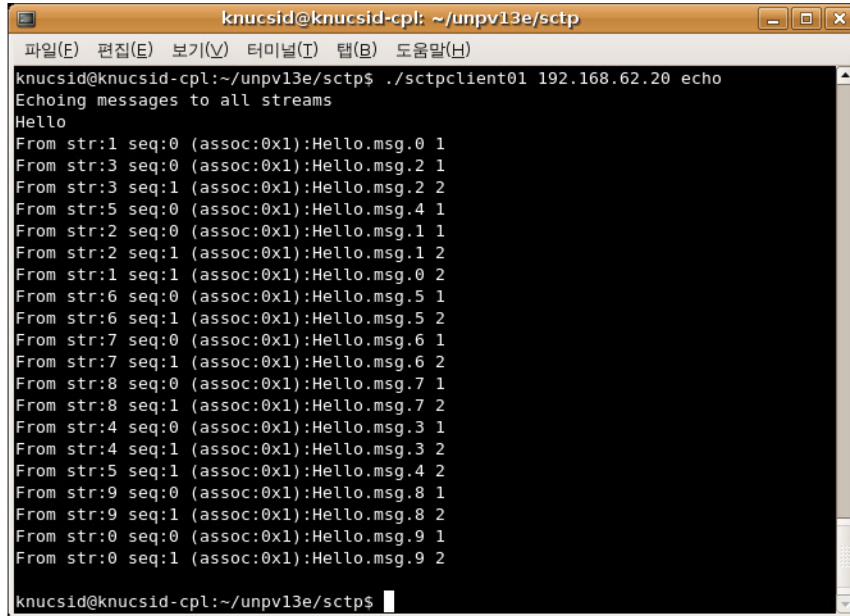
26-29 클라이언트는 그 번호를 1에서 2로 변경하고 이 업데이트된 메시지를 같은 스트림으로 전송한다.

Read messages and display

31 이 코드는 단지 하나의 작은 변경을 요구한다. 클라이언트는 에코 서버로부터 되돌아오는 메시지 수의 두 배로 한다.

Running the Modified Code

서버와 변경된 클라이언트를 이전처럼 시작하고, 클라이언트의 출력은 다음과 같다.



출력에서 볼 수 있는 것처럼 메시지가 손실되면 그 메시지가 손실된 특정 스트림만 지연된다. 다른 스트림들은 데이터 지연을 가지지 않는다. SCTP 스트림들은 연관된 메시지들의 집합에서 순서를 지키면서 head-of-line blocking을 피하는 강력한 메커니즘이다.

10.5 STREAM 수의 제어

SCTP 스트림을 어떻게 사용하는지는 봤지만, association 초기화에서 단말이 요구하는 스트림 수를 어떻게 제어할 것인가? 이전의 예제에서 outbound 스트림 수는 시스템 기본 설정으로 사용했다. FreeBSD KAME의 SCTP 구현에서는 기본적으로 10개의 스트림으로 설정한다. 응용이나 서버에서 10개 이상의 스트림을 사용하려면 어떻게 해야 하는가?

다음 그림에서 서버가 association이 시작할 때 단말이 요구하는 스트림 수를 증가시키는 것을 허락하도록 바꾸는 것을 보여준다. 이 변경은 association을 생성하기 전에 소켓상에서 해야만 한다.

```
sctp/sctpserv02.c

14     if( argc == 2 )
15         stream_increment = atoi(argv[1]);
16     sock_fd = Socket( AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP );
17     bzero( &initm, sizeof(initm) );
18     initm.sinit_num_ostreams = SERV_MORE_STRMS_SCTP;
19       Setsockopt(    sock_fd,    IPPROTO_SCTP,    SCTP_INITMSG,    &initm,
sizeof(initm) );
```

Initial setup

14-16 이전처럼 서버는 추가적인 매개변수를 바탕으로 플래그를 설정하고 소켓을 연다.

Modifying the streams request

17-19 이 라인은 서버에 추가하는 새로운 코드이다. 서버는 sctp_initmsg 구조체를 처음에 0으로 초기화한다. 이 변경은 setsockopt이 다른 어떤 값으로 무심코 변경되지 않을 것이라고 보증한다. 서버는 요구하는 스트림 수를 위해 sinit_max_ostreams 필드를 설정한다. 그 후 초기 메시지 매개변수로 소켓 옵션을 설정한다.

소켓 옵션을 설정하는 다른 방법은 sendmsg 함수를 사용하고 요구하는 다른 스트림 매개변수를 보조적인 데이터로 제공하는 것이다. 이 보조적인 데이터 방법은 단지 일대일(one-to-one) 소켓 인터페이스에서만 효과가 있다.

10.6 세션 종료의 제어

예제에서 association을 중단시키기 위해 클라이언트의 소켓 종료에만 의존해왔다. 그러나 클라이언트 응용은 항상 소켓을 종료하길 원하지 않는다. 그 같은 문제로 서버는 대응 메시지를 보낸 후에 association을 연 상태를 유지하길 원하지 않을 수도 있다. 이 경우 association을 중단하기 위한 또 다른 메커니즘을 필요로 한다. 일대다(one-to-many) 스타일 인터페이스에서는 응용에서 두 가지 방법이 가능하다. 하나는 우아한 방법이고 다른 하나는 그렇지 않다.

서버가 메시지를 보낸 후에 association을 중단하고자 하면 대응 메시지에 sctp_sndrcvinfo 구조체의 sinfo_flags 필드에 MSG_EOF 플래그를 적용한다. 이 플래그는 보낸 메시지에 응답을 한 후에 association의 중단을 강제한다. 다른 대안으로는 sinfo_flags 필드에 MSG_ABORT 플래그를 적용한다. 이 플래그는 ABORT 청크로 즉시 association의 종료를 강제한다. ABORT 청크는 TCP RST 세크먼트와 유사하고 지연 없이 어떤 association이라도 종료하게 한다. 아직 전송되지 않은 그 어떤 데이터라도 버려질 것이다. 그러나 ABORT 청크로 인한 SCTP 세션을 종료하는 것은 TCP의 TIME_WAIT 상태를 방지하는 것처럼 부정적인 측면을 가지지는 않는다. ABORT 청크는 우아한 abortive 종료를 하게 한다. 다음 그림은 상대방에게 대응 메시지를 보내졌을 때 에코 서버가 우아한 중단을 하기 위한 변경을 보여준다.

sctp/sctpserv03.c

```
25     for( ;; ) {
26         len = sizeof(struct sockaddr_in);
27         rd_sz = Sctp_recvmsg( sock_fd, readbuf, sizeof(readbuf),
28                               (SA *) &cliaddr, &len, &sri, &msg_flags );
29         if( stream_increment ) {
30             sri.sinfo_stream++;
31             if( sri.sinfo_stream >=
32                  sctp_get_no_strms(sock_fd, (SA *) &cliaddr, len) )
33                 sri.sinfo_stream = 0;
34         }
35         Sctp_sendmsg( sock_fd, readbuf, rd_sz,
36                      (SA *) &cliaddr, len,
37                       sri.sinfo_ppid,
38                        (sri.sinfo_flags | MSG_EOF), sri.sinfo_stream, 0,
0 );
```

<Server terminates an association on reply>

Send back responses, but shut down association

38 이 라인의 변화는 단순히 sctp_sendmsg 함수에서 MSG_EOF 플래그를 OR하는 것이다. 이 플래그 값은 응답 메시지를 성공적으로 전송한 후에 서버가 association을 종료하도록 한다.

```
sctp/sctpclient02.c

25     if( echo_to_all == 0 )
26      sctpstr_cli( stdin, sock_fd, (SA *)  &servaddr, sizeof(servaddr) );
27     else
28         sctpstr_cli_echoall( stdin, sock_fd, (SA *)  &servaddr,
29                         sizeof(servaddr) );
30     strcpy(byemsg, "goodbye" );
31     Sctp_sendmsg( sock_fd, byemsg, strlen(byemsg),
32              (SA *) &servaddr, sizeof(servaddr), 0, MSG_ABORT, 0, 0, 0 );
33     Close(sock_fd);
```

<Client aborts the association before closing>

Abort association before close

30-32 클라이언트는 사용자 오류 원인처럼 중단을 포함하는 메시지를 준비한다. 클라이언트는 MSG_ABORT 플래그를 가지는 sctp_sendmsg 함수를 호출한다. 이 플래그는 ABORT 청크로 전송하고 즉시 association은 종료된다. ABORT 청크는 상위 이유 필드에서 "goodbye"와 같은 메시지와 사용자 초기화된 오류 원인을 포함한다.

Close socket descriptor

33 이미 association은 중단되었지만 associated 된 시스템 리소스를 해제하기 위해 socket descriptor를 종료해야 한다.

부록 A. TCP/IP 인터넷 프로토콜

A.1 인터넷 서비스

우리가 쉽게 접할 수 있는 주요 인터넷 서비스로는 다음이 있다.

- 월드와이드웹
- 인터넷 검색
- 인터넷 카페
- 온라인 게임
- 메신저
- 전자메일
- 홈페이지
- 블로그
- 위키

A.1.1 월드와이드웹

월드와이드웹(WWW; World Wide Web)이란 '세계 규모의 거미집 모양의 망'이라는 뜻으로, '하이퍼텍스트' 기능 혹은 'HTTP(HyperText Transfer Protocol)' 프로토콜을 사용하여 인터넷상에 분산되어 존재하는 온갖 종류의 정보를 통일된 방법으로 찾아볼 수 있게 하는 서비스 혹은 소프트웨어를 의미한다. 다음 그림은 WWW 의 예로써 '네이버' 웹페이지의 초기 화면을 보여준다.

<그림 A.1> 웹페이지

월드와이드웹은 줄여서 '웹(web)'이라고 부르며 최근 웹서비스가 주목 받고 있는 이유는, 문자 정보가 대부분이었던 과거의 통신수단과는 달리 WWW 에서는 문자, 화상, 음성에 등의 다양한 데이터 표현 및 전달이 가능하기 때문이다. WWW 의 주요 특징은 다음과 같다.

(1) 웹문서는 웹서버(web server)라고 하는 컴퓨터에 '하이퍼텍스트' 형식으로 저장되어 있다.

(2) 웹문서를 보기 위하여 우리는 웹브라우저(web browser)라는 소프트웨어가 필요하다.

(3) 대표적인 웹브라우저로는 인터넷 익스플로러(Internet Explorer) 등이 있다.

하이퍼텍스트란 링크(link)에 의해서 상호연결된 텍스트 혹은 문서를 의미하며, 인터넷에 분산되어 있는 세계의 웹문서는 하이퍼텍스트로 연결될 수 있다. 전 세계의 하이퍼텍스트가 이리저리로 연결된 모습이 마치 거미가 집을 지은 것처럼 보이기 때문에 '월드와이드웹'이라는 이름이 붙여졌다. HTTP 는 하이퍼텍스트 웹문서를 전달하기 위해 사용되는 프로토콜(규칙)을 의미한다.

예) http://www.empas.com/

A.1.2 홈페이지

홈페이지란 개인이나 기관에서 자신의 홍보 및 소개를 위해 제작한 웹사이트 혹은 웹문서를 의미한다. 대부분의 학교, 기업, 정부기관 등에서 홍보를 위해 자체 홈페이지를 구축해 놓고 있다. 또한, 각 개인도 자신 소개 및 홍보를 위하여 '개인 홈페이지'를 가지고 있다. 다음 그림은 교육과학기술부의 홈페이지 화면이다.

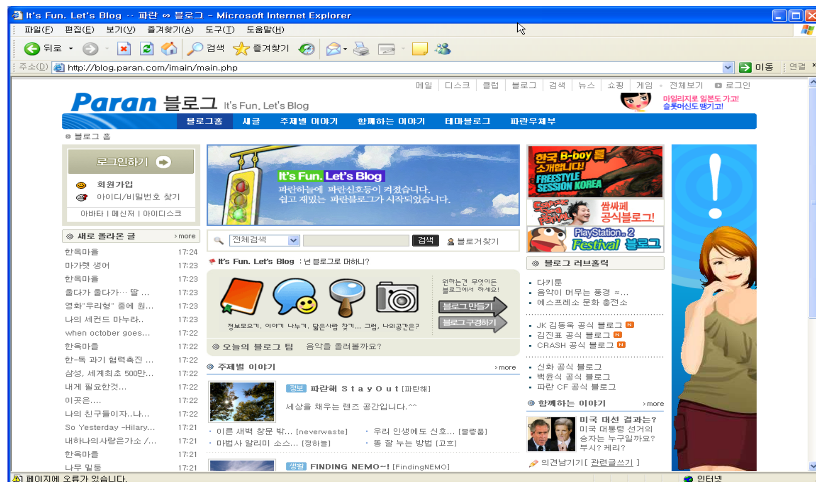
<그림 A.2> 교육과학기술부 홈페이지

홈페이지 제작에 사용되는 프로그래밍 언어를 HTML(HyperText Markup Language)라한다. 홈페이지를 편리하게 저작하는 시각 프로그램인 소프트웨어로 '나모 웹 에디터' 등의 편집기가 있다. 실제 홈페이지 제작을 위해서는 HTML 외에도 JavaScript, ASP, JSP, PHP 등의 웹 프로그래밍 언어가 필요하며, 게시판 기능을 위해 MySQL 등을 이용한 데이터베이스 구축도 필요하다. 휴대 단말용' 무선인터넷 웹페이지도 제작할 수 있는데 m-HTML 이나 WML (Wireless Markup Language) 등의 웹 프로그래밍 언어를 사용한다.

A.1.3 블로그와 위키

블로그는 홈페이지와 유사한 것으로 '웹(web)'과 항해일지를 뜻하는 '로그(log)'의 합성어로 "인터넷 일기" 혹은 "인터넷 항해일지"를 의미한다. 블로그는 주인인 블로거가 저자이며 미디어 생성자이다. '한국형 블로그'를 '미니 홈페이지'라 부르기도 한다.

최근 인터넷 포털사이트에서 블로그 서비스를 제공하는 네이버블로그, 엠파스블로그, 파란블로그 등이 있다. 또한 최근 개인 홈페이지는 블로그 서비스로 제공되고 있으며, 휴대폰 등의 단말에서 제공되는 모블로그는 mobile 과 blog 의 합성어이다. 다음 그림은 파란블로그의 웹페이지 화면이다.



<그림 A.3> 블로그

위키란 최근에 등장한 인터넷 서비스로서 홈페이지 및 블로그와 유사한 성격을 가지나, 누구나 문서의 내용을 자유로이 추가, 변경 등의 편집을 할 수 있다는 측면에서 기존의 웹페이지와는 구별된다. 위키(WiKi)라는 용어는 하와이 원주민 말로 "빨리"라는 뜻을 지닌 "위키위키"에서 나온 말이며, "WiKi = What I Know Is" 표현의 줄임말이라는 견해도 있다. 위키페이지는 특히 과학기술 전문사이트에서 많이 사용되고 있으며, 유사한 주제를 연구하는 전문가들끼리 전문지식을 공유하고 최근 정보를 교환하는 토론의 장으로 널리 활용되고 있다.

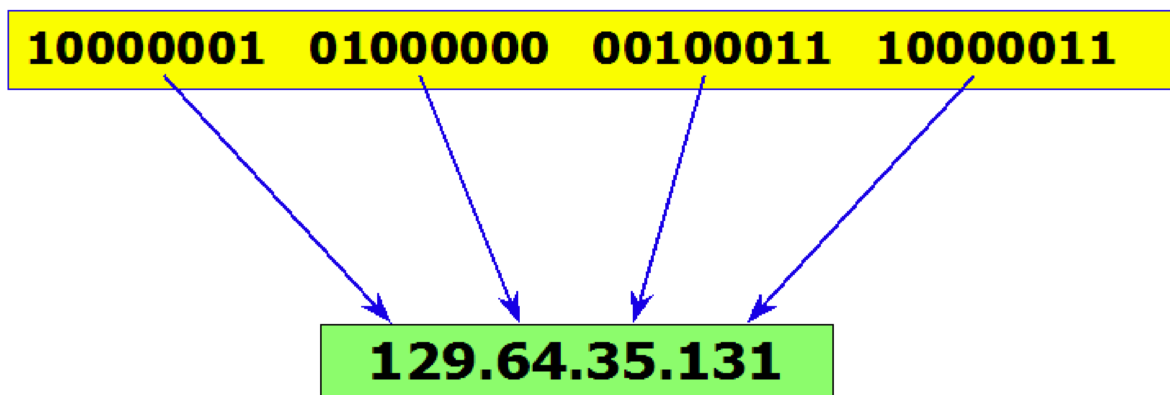
다음 그림은 위키 관련 사이트인 '위키피디아'의 웹사이트이다.

<그림 A.4> 위키사이트(http://www.wikipedia.org/)

A.2 인터넷 주소와 도메인 이름

A.2.1 인터넷 주소

인터넷 주소 혹은 IP 주소란 인터넷 상에서 내 컴퓨터를 식별하기위한 주소이다. 즉, IP 주소를 통해 상대방의 컴퓨터에 인터넷 데이터를 보낼 수 있으며, 또한 상대방이 내 컴퓨터에 데이터를 전달할 수 있다. IP 주소는 "129.64.35.131"처럼 4 개의 십진수와 "."를 사용하여 표현한다.



<그림 A.5> IP 주소의 표현

전 세계의 인터넷 주소 할당을 관리하는 기관은 미국에 있는 IANA(Internet Assigned Numbers Authority)이다. IANA 에서는 각 국가 혹인 단체에게 IP 주소를 할당 및 분배하는 총괄 업무를 관장하고 있다. 세계적으로 IP 주소할당을 효과적으로 추진하기 위하여 지역별로 IANA 의 역할을 대행하는 기관이 있는데 아시아-태평양 지역은 APNIC(Asia Pacific Network Information Center)에서 IP 주소 할당을 수행하고 있다. 또한 국가별로 주소를 할당해 주는 기관이 있는데 한국은 KRNIC 에서 주소 할당 업무를 수행하고 있다. IP 주소를 받고 싶은 개인이나 단체는 KRNIC 에 주소할당을 요청하거나 APNIC 이나 IANA 에 직접 주소를 요청할 수도 있다.

<그림 A.6> IP 주소할당 기관

A.2.2 도메인 이름과 DNS

도메인 이름이란 한마디로 컴퓨터의 이름이다. 즉, 인터넷에서 컴퓨터를 식별하기 위해 사용하는 컴퓨터의 고유 이름이다. 예를 들어, 웹이나 인터넷 서비스에서 사용하는 "www.naver.com" 는 도메인 이름이다.

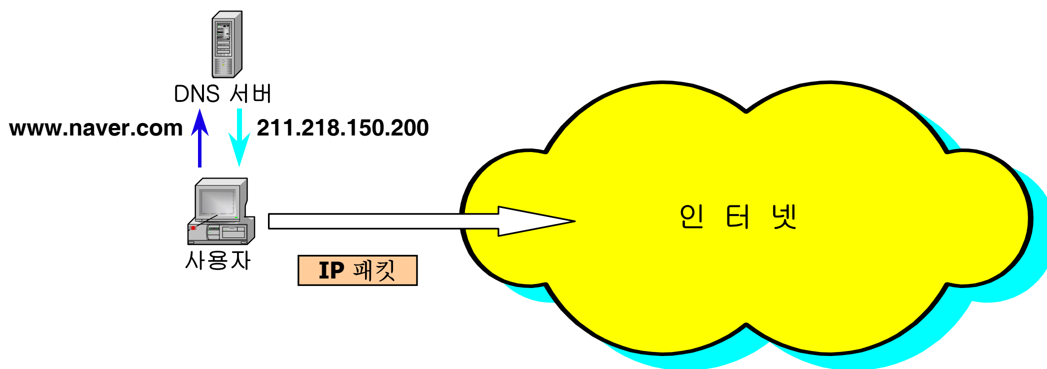
엄밀히 얘기하면 도메인 이름과 컴퓨터 이름은 다르다. 예를 들어, www.naver.com 은 naver.com 도메인에 있는 웹서버(컴퓨터)의 이름이고, naver.com 은 도메인 이름이다. 즉, naver.com 도메인은 웹서버 (www.naver.com) 이외에도 다른 많은 컴퓨터를 포함할 수 있다(예: mail.naver.com).

앞에서 "인터넷에서 컴퓨터를 식별하기 위한 용도로써" IP 주소를 얘기하였다. IP 주소처럼 도메인 이름도 컴퓨터의 식별자로 사용한다. 즉, 도메인 이름은 IP 주소로 대응한다. 예를 들어, "www.naver.com" 도메인 이름은 "211.218.150.200"의 IP 주소를 갖는다.

그렇다면, 왜 IP 주소 외에 도메인 이름을 사용할까? IP 주소는 사람이 기억하고 사용하기에 불편하다. 예를 들어, 브라우저의 검색창에 "211.218.150.200" 이라는 IP 주소를 외워서 입력하기 보다는, www.naver.com" 이라는 웹서버의 이름을 입력하는 게 훨씬 더 편리하다.

도메인 이름 시스템 서버란 이미 언급한 "도메인이나 컴퓨터 이름을 IP 주소로 바꾸어 주는 서버"이다. 앞에서 "TCP/IP 인터넷 설정" 과정에서 'DNS(Domain Name System) 서버'의 주소를 설정하였다. 바로 여기에 설정한 DNS 서버가 검색창에 입력하는 컴퓨터 이름을 IP 주소로 바꾸어 준다.

다음 그림은 웹브라우저를 사용하는 경우, DNS 와의 동작절차를 보인다.



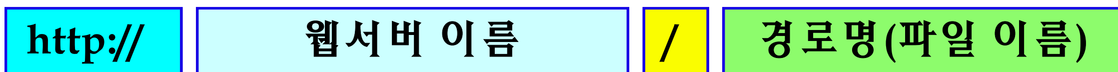
<그림 A.7> DNS 서버

그림에서 웹브라우저의 주소란에 www.naver.com"이라 입력하면 컴퓨터는 미리 설정된 DNS 서버에게 조회하여 해당하는 IP 주소가 무엇인지 알아온다. DNS 서버가 해당하는 IP 주소를 웹 브라우저에게 알려주면 웹 브라우저는 '해당 IP 주소'를 사용하여 웹서버에게 인터넷 패킷을 전송한다.

보통은 각 학교나 회사 등의 기관별로 DNS 서버를 운영하고 있으며, 인터넷의 도메인 이름은 계층적으로 구성되어 있다. 예를 들어, www.mest.go.kr"(교육과학기술부 홈페이지)의 경우, 'kr'은 한국(KOREA)을 의미하고, 'go'는 정부기관(government)을 'most'는 교육과학기술부를 의미한다. '계층적'이란 의미는 'kr' 도메인 아래에 'go', 'co' (company), 'ac' (academy) 등의 하위 도메인이 있고, 또한, 'go' 도메인 아래에 'mest'(교육과학기술부) 등의 여러 하위 도메인이 있다는 것을 뜻한다.

A.2.3 웹주소

이제 도메인 이름을 사용하는 웹주소에 대하여 살펴본다. 웹주소란 URL(Uniform Resource Locator)이라고도 하며, 인터넷에서 특정 '웹문서의 위치'를 나타내는 주소이다. 웹주소는 대개 "컴퓨터(웹서버) 이름"과 "경로명"으로 구성된다. 다음 그림은 웹주소 형식이다.



<그림 A.8> 웹 주소 형식

예를 들어, "www.naver.com/index.html"은 네이버 웹서버(www.naver.com)에서 "index.html"이란 문서 경로를 나타내는 URL 이다. 즉, 웹브라우저의 주소창에 "www.naver.com/index.html"라고 입력하면 해당 문서가 화면에 출력된다. 실제로는 "www.naver.com"라고 입력해도 자동으로 "www.naver.com/index.html" 문서를 참조하도록 되어 있다.

최근에는 '인터넷의 한글화' 추세에 따라 '한글' 도메인 이름도 사용하고 있다. 즉, 각 기관의 홈페이지 주소를 영문명 대신에 한글명으로 입력해도 해당 홈페이지에 접속할 수 있다. 다음 그림은 '청와대' 홈페이지의 영문 도메인명인 www.president.or.kr 대신에 한글 도메인명인 '청와대'를 입력한 경우의 화면이다.

<그림 A.9> 한글 도메인 사용 예

해당 홈페이지를 한글 도메인명으로 접속하려면 해당 기관에서 관련 한글 도메인명 주소를 도메인 관리기관에 등록해야 한다. 국내에서 도메인명 등록은 '한국인터넷진흥원(NIDA)'을 통하여 이루어지고 있다. 관련 웹주소는 http://www.nida.or.kr 혹은 http://domain.nic.or.kr 이다.

A.3 모바일 인터넷

지금까지 유선인터넷에서 사용하는 여러 인터넷 서비스에 대하여 살펴보았다. 최근 휴대폰이나 PDA 등의 무선 휴대단말에서 인터넷을 사용하는 '모바일 인터넷' 혹은 '무선 인터넷' 서비스에 대하여 살펴본다.

A.3.1 모바일 인터넷 서비스

모바일이나 무선 인터넷은 이동/무선 휴대 단말로 언제 어디서나 인터넷에 접속하여 다양한 정보검색과 전자상거래 등의 섭스 제공을 의미한다. 즉, 기존 인터넷 환경의 시간적 공간적 제약을 극복한 유비쿼터스 인터넷 서비스 환경이다. 휴대폰에서 음성전화 서비스만 제공하였으나, 모바일 인터넷 서비스로 인해 "유선과 무선 기술이 통합"되고 "PC 와 휴대폰 서비스가 융합"되었다.

모바일 인터넷에서는 기존 유선 인터넷에서 제공되던 웹서핑, 게임, 메신저 및 온라인 전자상거래를 포함하여 사실상 모든 종류의 인터넷 서비스를 무선 휴대단말기를 통하여 제공한다. 다음 그림은 휴대폰에서 사용할 수 있는 모바일 인터넷 서비스 화면의 예이다.

<그림 A.10> 모바일 인터넷 서비스

A.3.2 모바일 인터넷 기술

유선 인터넷과는 달리 모바일 인터넷은 단말기 화면의 크기가 작고, 네트웍 전송 속도가 느린 특징이 있다. 따라서, 유선인터넷에서 사용하던 HTTP, HTML 등의 기술을 무선 인터넷 환경에 그대로 적용하기는 어렵다. 그러므로 모바일 인터넷 서비스를 위해서는 전용 웹브라우저와 전용 마크업 언어가 필요하다.

모바일 인터넷용 웹브라우저에는 WAP(Wireless Application Protocol) 브라우저와 ME(Mobile Explorer)가 많이 사용된다. WAP 브라우저는 "WAP 포럼(www.wapforum.org)"이라는 국제 표준기구에서 개발한 것이고, ME 는 Microsoft 사에서 모바일용으로 개발한 브라우저이다. 또한, 마크업언어 관점에서도 기존 유선 인터넷 환경에서 사용하던 HTML 대신에 WML(Wireless Markup Language) 혹은 m-HTML(mobile HTML) 등의 언어를 사용하여 웹문서가 제작되어야 한다.

다음은 국내 이동통신사업자들이 주로 사용하고 있는 브라우저와 마크업 언어이다.

이동통신사	대표 서비스명	브라우저	마크업 언어
SKT	NATE (www.nate.com)	WAP	WML
KTF	매직앤 (www.magicn.com)	ME	m-HTML
LGT	EZ-i (www.ez-i.co.kr)	WAP	WML

모바일 인터넷 서비스를 이용하기 위해서는 휴대폰, PDA 등의 휴대단말기에 WAP, ME 등의 모바일 전용 브라우저가 탑재되어 있어야 하며, 또한 웹서버에 위치하는 각종 웹문서들도 WML, m-HTML 등의 전용 마크업 언어로 작성되어야 한다.

모바일 인터넷 서비스를 위한 무선 접속기술은 cdma2000 기술이 널리 사용되고 있으며, 최근에는 '와이브로(WiBro)'라 불리는 차세대 무선접속 기술이 등장하였다. 한국에서는 다양한 모바일 인터넷용 서비스 및 컨텐츠의 효율적인 개발을 위해 WIPI(Wireless Internet Platform for Interoperability)라는 무선인터넷 플랫폼 기술을 표준화하였다(www.wipi.or.kr).

A.3.3 모바일 인터넷 주소

한편, 휴대단말에서 무선인터넷을 사용할 수 있도록 모바일주소(WINC) 서비스도 제공되고 있다. 모바일주소란 무선인터넷 이용자들의 이용환경을 개선시키기 위하여 국가 인터넷 주소자원 관리기관인 한국인터넷진흥원(NIDA)에서 국내 이동통신사와 공동으로 복잡한 URL 대신 번호를 통해 무선인터넷에 접속하도록 실시하는 공공서비스이다.

예를 들어, 한국인터넷진흥원 도메인은 www.nida.or.kr 인데, 여기서 www 와 최상위도메인을 제거한 문자는 nida 이며, 각 글자에 해당하는 휴대폰 키패드의 숫자(번호)는 6432 이다. 또한 고유번호는 0 이므로, 한국인터넷진흥원의 모바일주소(WINC)는 6432#0 이다. 이처럼 도메인 문자들을 숫자로 대응시키는 과정에서 중복이 발생할 수 있으므로, 실제로는 좀 더 크고 다양한 WINC 번호가 할당된다.

<그림 A.11> 모바일 주소 예

<메모>

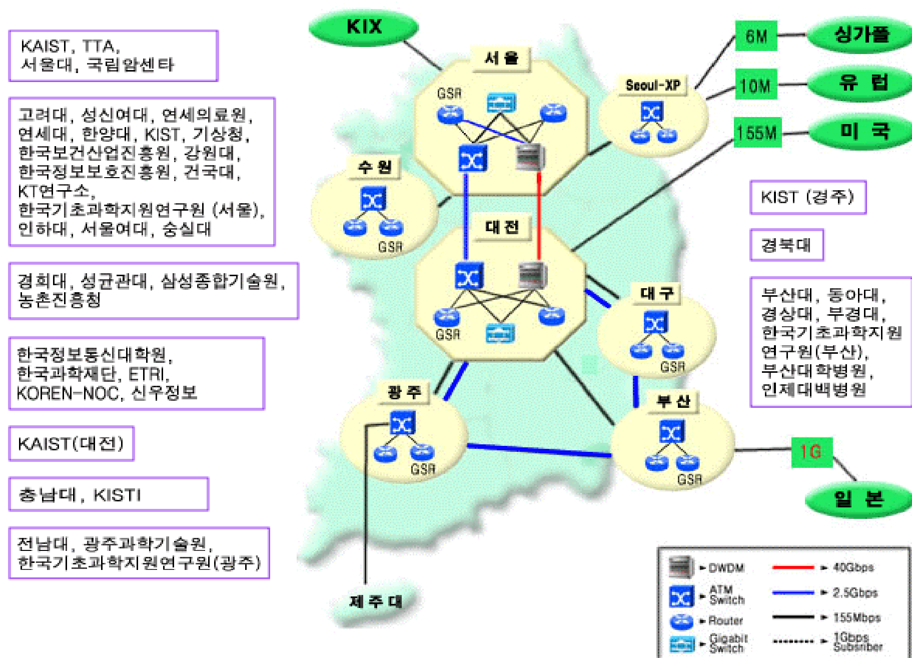
- WINC 는 Wireless Internet Numbers for Contents 의 약어로써, "무선인터넷 콘텐츠 접근번호체계"를 의미함
- 보다 상세한 WINC 사용법은 다음 홈페이지 참조 (http://www.winc.or.kr/)
- 실제로 휴대폰으로 버스 위치를 검색해는 "버스정보시스템(businfo)"에서 WINC 번호를 사용하고 있음

A.4 다양한 인터넷 통신망

인터넷 통신망 혹은 컴퓨터 네트워크란 말 그대로 컴퓨터로 구성되는 네트워크이며 사실상 인터넷을 의미한다. 즉, 인터넷 서비스는 인터넷이라는 컴퓨터 네트워크를 통해 이루어지는 컴퓨터 통신서비스인 셈이다. 여기서는 인터넷과 관련된 다양한 통신망에 대하여 살펴본다.

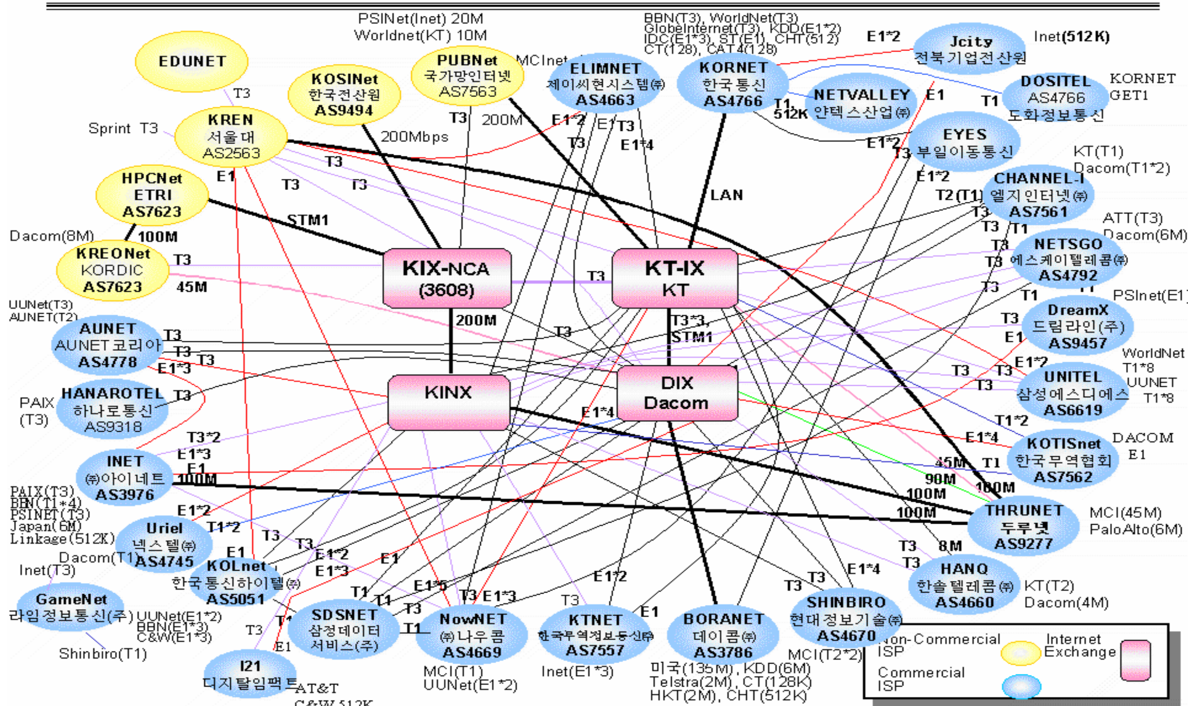
A.4.1 인터넷

인터넷은 1970 년대 초에 미국 대학 및 연구소를 중심으로 컴퓨터간 데이터 통신을 위해 처음으로 인터넷이 개발되었다. 국내에 인터넷이 널리 보급된 시기는 1990 년대에 WWW 이 활성화되면서 본격적으로 인터넷이 사용되기 시작하였다. 다음 그림은 국내 인터넷 선도시험망 구성도이다.



<그림 A.12> 국내 인터넷 선도시험망

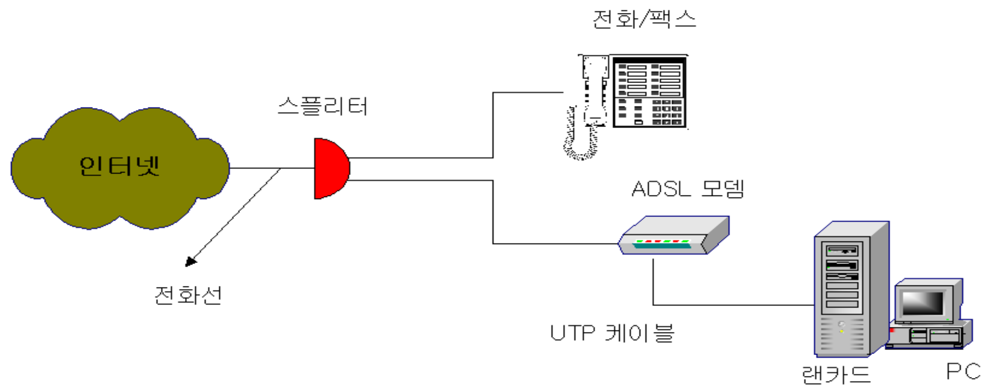
실제 인터넷은 여러 가지 다양한 종류의 네트워크의 결합체이다. 인터넷서비스사업자(ISP)별로 소규모 인터넷이 구성되고, 이러한 소규모 인터넷이 국내 인터넷 백본망으로 결합되어 해외에 있는 인터넷과도 연결된다. 다음 그림은 실제 국내 인터넷 상용망 구성도를 이다.



<그림 A.13> 국내 상용 인터넷망

A.4.2 DSL

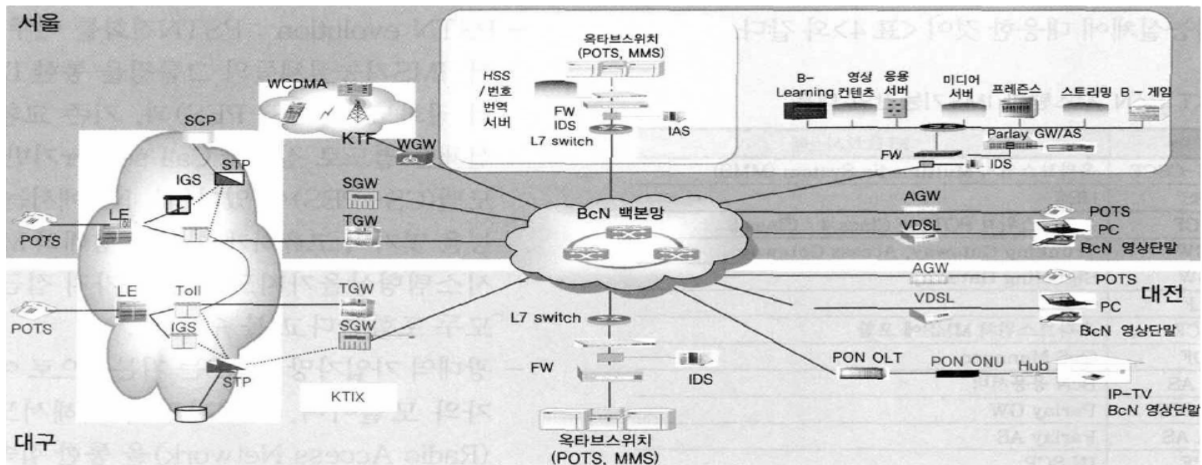
우리 가정에서 사용하는 PC 는 대개 DSL(Digital Subscriber Line)이라는 접속기술을 사용하여 인터넷에 접속하게 된다. 특히, ADSL(Asymmetric DSL) 접속기술이 널리 사용되고 있는데, 이는 각 가정의 PSTN 전화회선을 사용하여 인터넷에 접속할 때에 사용되는 기술이다. 각 가정에 유입되는 전화회선은 스플리터(splitter)를 통해 일반전화용과 ADSL 용으로 분리되고, PC 는 ADSL 모뎀을 사용하여 인터넷에 접속된다.



<그림 A.14> ADSL 기반 인터넷 접속

A.4.3 BcN

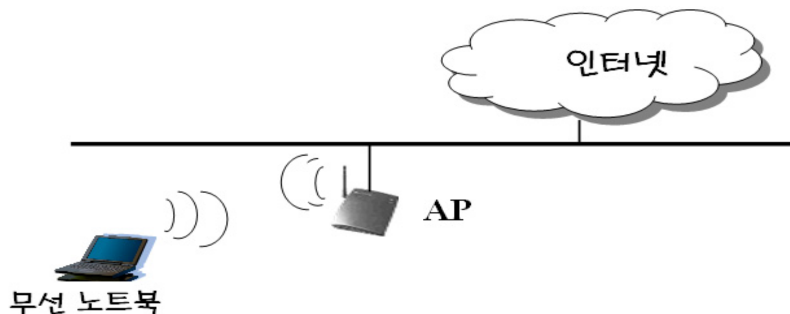
인터넷 사용자가 급증하고 또한 기존의 인터넷 서비스 뿐만 아니라 각종 이미지, 오디오, 비디오 등의 멀티미디어 데이터가 인터넷을 통해 제공됨에 따라, 인터넷 망으로 광대역 통신망으로 개선시킬 필요성이 대두 되었다. 이와 같은 인터넷망의 고도화 정책에 따라 국내에서는 차세대 통신망이라 불리는 BcN(Broadband convergence Network) 망기술을 개발하게 되었다. 다음 그림은 KT 에서 구축중인 BcN 망 구성도이다.



<그림 A.15> 국내 BcN 망 구성도

A.4.4 무선랜

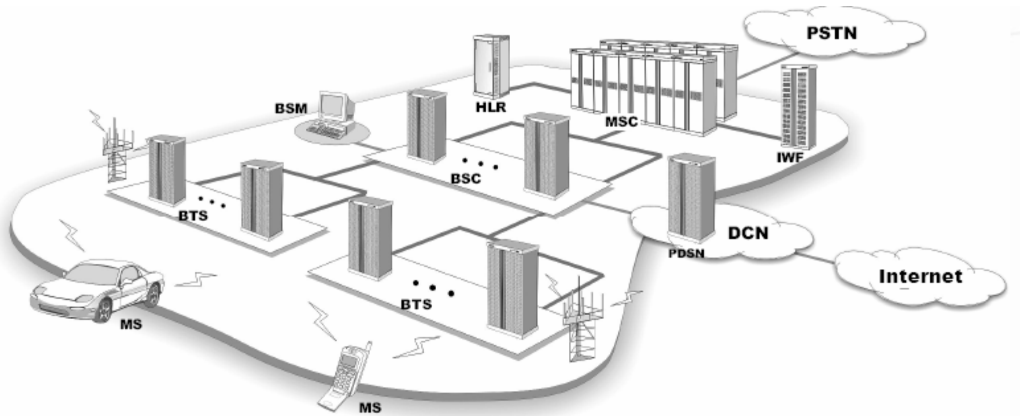
무선랜 혹은 WLAN 기술은 LAN 기술을 무선 영역으로 확대한 기술로써, 노트북 등의 이동 단말장치에 무선랜 카드를 장착하여 인터넷 접속에 사용되는 기술이다. 무선랜 접속을 위해서는 무선랜카드와 엑세스 포인트라는 장비가 필요하다. 노트북에 무선랜카드를 장착하고, 주변에 설치된 AP 를 통해 인터넷에 접속하게 된다. 다음 그림은 간략한 무선랜망 구성도이다.



<그림 A.16> 무선랜 망 구성도

A.4.5 이동통신망

우리가 사용하는 휴대폰은 이동통신망을 사용하여 데이터 통신을 수행하는데, 현재 사용되는 기술을 이동통신기술 혹은 IMT-2000(International Mobile Telecommunications - 2000) 기술이라 한다. 특히, IMT-2000 기술은 휴대폰에서 이동전화 뿐만 아니라 무선인터넷 서비스도 제공하고 있는데, 세부적인 무선접속기술에 따라 크게 CDMA(Code Division Multiple Access) 및 W-CDMA (Wideband CDMA) 기술로 분류할 수 있다. 다음 그림은 우리가 흔히 사용하는 CDMA2000 이동통신망의 구성도이다.



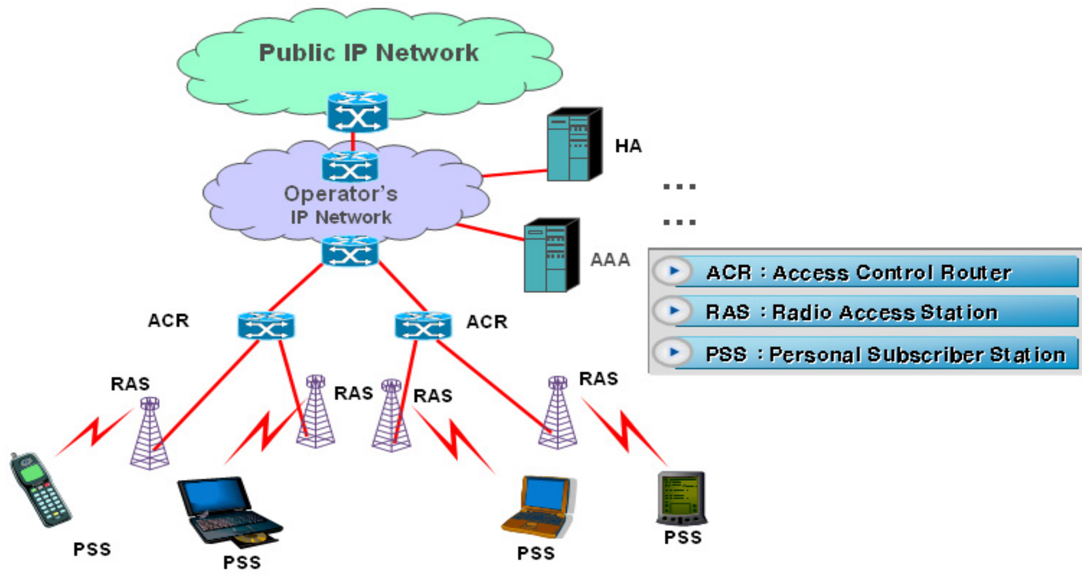
<그림 A.17> 이동통신망 구성도

A.4.6 와이브로

한편, 무선 이동통신망을 사용한 인터넷 접속의 광대역화 및 이동성 제공을 위해 새로이 등장한 무선접속기술이 있는데, 바로 와이브로(WiBro: Wireless Broadband) 기술이다. 와이브로 기술은 차세대 무선통신기술로 불리며, 기본 WLAN 커버리지를 1 km 대로 확대하였으며, 이동 중에도 유연한 무선 인터넷 서비스 제공을 주요 목표로 하고 있다.

기존 휴대폰의 이동통신기술이 음성전화서비스와 인터넷서비스를 동시에 제공하는 것과는 달리, 와이브로는 인터넷 서비스에 특화된 기술이며, 높은 데이터 수신 속도와 비교적 저렴한 이용요금을 특징으로 한다.

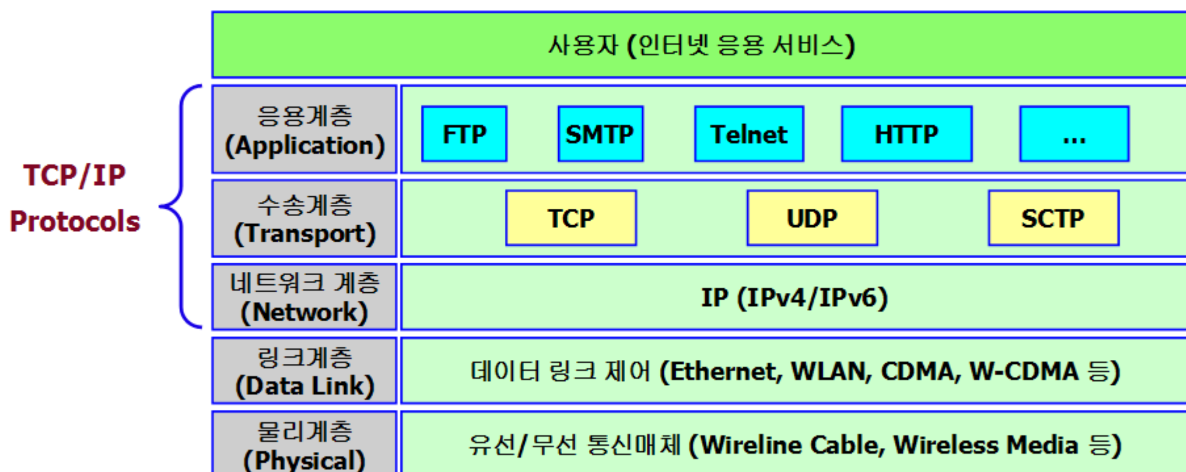
다음 그림은 와이브로 서비스를 위한 네트워크 구성도이다. 이동단말은 RAS 기지국에 연결되며, ACR 장비를 통해 인터넷에 접속된다.



<그림 A.18> 와이브로 통신망

A.5 프로토콜 스택

인터넷(Internet) 통신은 다음 그림처럼 'TCP/IP 프로토콜(protocol) 스택(stack)'에 해당하는 여러 가지 프로토콜들을 사용하여 이루어진다.



<그림 A.19> TCP/IP 프로토콜 스택

TCP/IP 프로토콜 스택은 기존의 OSI(Open System Interconnection) 7 계층 모델과는 달리 물리계층, 링크계층, 네트워크 계층, 수송계층 및 응용계층으로 나뉘어진다. 그림에서 응용계층은 OSI 모델의 계층 5(세션계층), 계층 5(표현계층), 계층 7(응용계층)의 기능을 모두 제공한다. 예를 들면, FTP 는 파일전송과 관련된 세션제어, 표현 및 응용 기능 모두를 제공한다.

특히, 네트워크 계층의 IP 프로토콜과 수송계층의 여러 프로토콜, 그리고 응용계층의 여러 응용 프로토콜들을 TCP/IP 프로토콜 스택이라 부른다. 이와 같은 TCP/IP 프로토콜 스택은 하위 링크계층 및 물리계층에 다양한 종류의 프로토콜 혹은 접속기술을 지원한다.

<메모>

- 프로토콜이란 한 마디로 "통신규약 혹은 통신규칙"을 의미한다. 즉, 두 대의 컴퓨터가 서로 통신을 하기위해서는 통신 프로토콜에 따라 동작을 해야 한다. 따라서 프로토콜은 모든 인터넷 장비들이 지켜져야 하며, 대개 표준화 기구에서 통신용 프로토콜 표준을 제정한다.

- TCP/IP 프로토콜의 표준화는 IETF(Internet Engineering Task Force) 표준화기구에서 표준으로 제정된다. (http://www.ietf.org/ 참조)

- 반면에 하위 링크 계층 및 물리계층 기술은 다양한 인터넷 접속(access) 기술을 의미하며, CDMA (http://www.3gpp.org/ 참조) 및 W-CDMA (http://www.3gpp2.org/ 참조) 등의 이동통신기술 뿐만 아니라, IEEE 802 위원회에서 표준으로 제정된 WLAN, WiBro 등의 다양한 무선접속기술을 포함한다. (http://www.ieee802.org/ 참조)

A.6 Internet Protocol (IP)

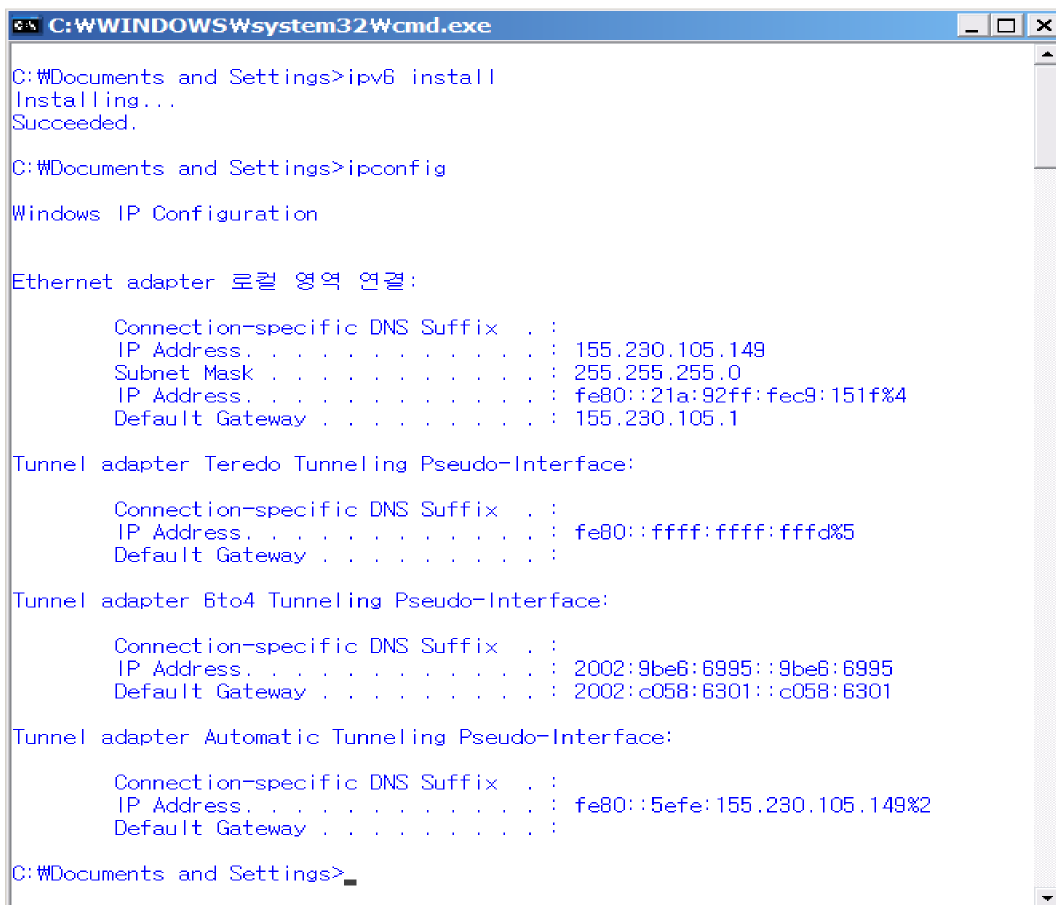
A.6.1 IP 주소의 표현

IP 프로토콜은 인터넷에서 컴퓨터간에 데이터 패킷(packet) 전달을 위해 사용되는데, 이를 위해서는 인터넷상의 특정 컴퓨터를 식별하기 위한 IP 주소(address)가 필요하다. 즉, IP 프로토콜은 해당 컴퓨터의 IP 주소를 참조하여 패킷을 최종 목적지에 전달한다.

IPv4(IP 버전 4)의 주소는 "129.64.35.131"처럼 4 개의 십진수로 표현되며, 이를 "dotted-decimal 표현방식"이라 한다. 각 숫자는 1byte(8bits)의 크기를 가지므로 "0~255" 사이의 숫자를 가지게 된다.

한편, IPv4 주소는 4 바이트(32 비트)의 크기를 가지는 반면에, IPv6 주소는 16 바이트(128 비트)의 크기를 갖는다. "3ffe:fffe::2085"처럼 IPv6 주소는 16 진수로 표현되고, 16 진수 4 개당 콜론(:)으로 구분하며, 콜론이 연속으로 2 개가 있는 경우에는 가운데에 모두 '0'이 있음을 의미한다.

다음 그림은 Window 에서 DOS 창을 통해 현재 사용 중인 IP 주소 정보를 파악하는 화면이다. Window XP 에서는 "ipv6 install" 명령을 통해 IPv6 기능을 사용할 수 있으며, 그림에서 해당 컴퓨터는 IPv4 주소와 함께 "2002:"로 시작하는 IPv6 주소를 사용하고 있음을 확인할 수 있다.



<그림 A.20> IP 주소 확인

A.6.2 IP 패킷 전달

TCP/IP 프로토콜의 주요 기능은 “인터넷을 통한 사용자 컴퓨터간 패킷(데이터) 전달”이다. 즉, 송신 컴퓨터의 데이터를 인터넷을 통하여 수신 컴퓨터에게 전달하는 것이다. TCP/IP 프로토콜에 의한 송수신 컴퓨터간 인터넷 패킷 전달과정을 정리하면 다음과 같다.

(1) 송신자는 전송할 응용 데이터를 만들고, 수신 컴퓨터의 IP 주소를 파악한다.

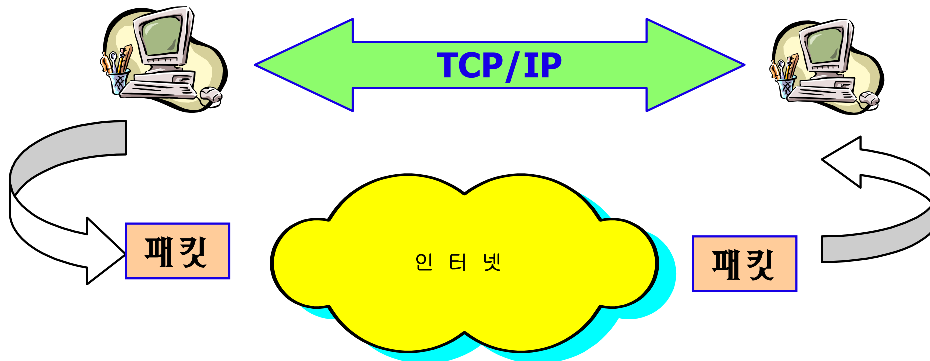
(2) IP 패킷 헤더에 수신자의 IP 주소를 첨가한 후, 헤더와 데이터로 구성되는 패킷을 만든다.

(3) 패킷을 인터넷으로 전송한다.

(4) 인터넷은 IP 라우팅 절차에 따라 수신자 컴퓨터에게 패킷을 전달한다.

(5) 수신 컴퓨터는 패킷에서 응용 데이터를 추출하여 상위에 있는 수신자에게 전달한다.

다음 그림은 TCP/IP 프로토콜을 이용한 IP 패킷 전달 과정이다.



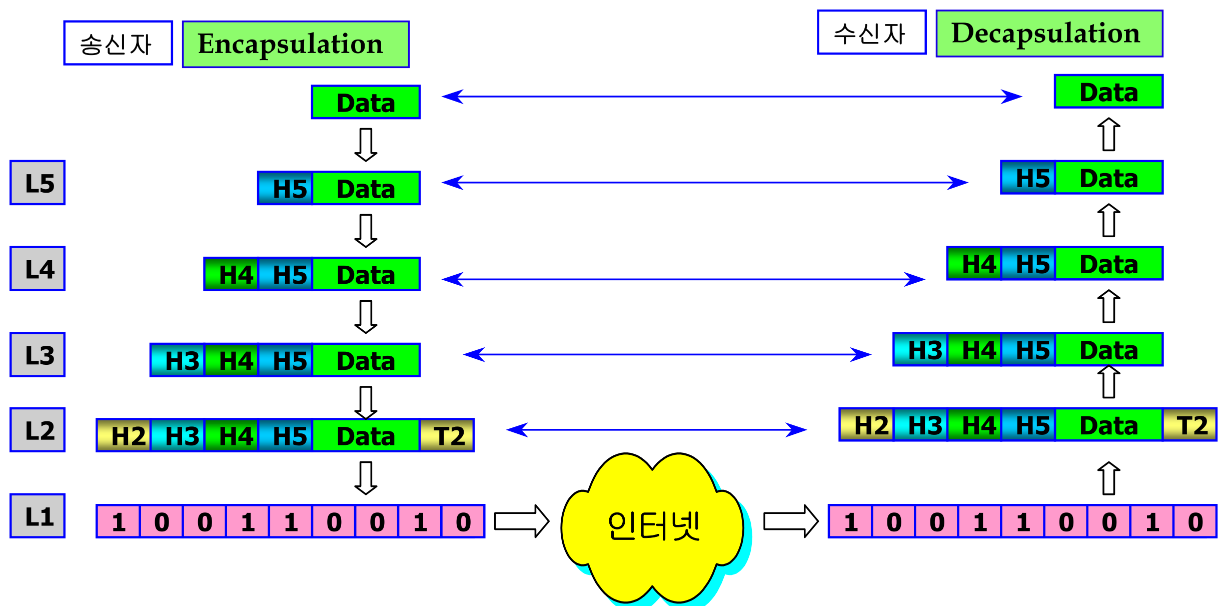
<그림 A.21> IP 패킷 전달

송신 컴퓨터에서 전송된 IP 패킷은 실제 인터넷 망에서 여러 개의 라우터를 거쳐 최종 수신 컴퓨터에 전달되는데, 이를 IP 라우팅이라 하며 IP 라우팅을 위해서 각 라우터들은 라우팅 테이블을 갖추고 있다.

A.6.3 패킷 캡슐화

TCP/IP 패킷 전달과정에서 중요한 점은 캡슐화(encapsulation)이다. 사용자의 데이터는 캡슐화 과정을 통해 인터넷 패킷으로 구성되고, 송신 컴퓨터를 떠나 인터넷을 향해한다. 인터넷을 도착한 패킷이 수신 컴퓨터에 도달했을 때, 데이터는 역캡슐화(decapsulation)를 통해 수신자에게 전달된다.

다음 그림은 TCP/IP 인터넷 패킷 전달을 위한 캡슐화 및 역캡슐화 과정이다. 그림에서처럼, 송신 컴퓨터가 발생시킨 데이터는 TCP/IP 프로토콜 스택을 따라 내려오면서, 각 계층에 해당하는 헤더 혹은 트레일러가 첨가된다. 이러한 헤더들의 정보는 각 TCP/IP 계층에서 역할 수행을 위한 중요한 정보를 지니고 있다. 예를 들어, IP 헤더에는 IP 주소가 포함되어 인터넷 패킷전달에 중요한 정보로 사용된다.

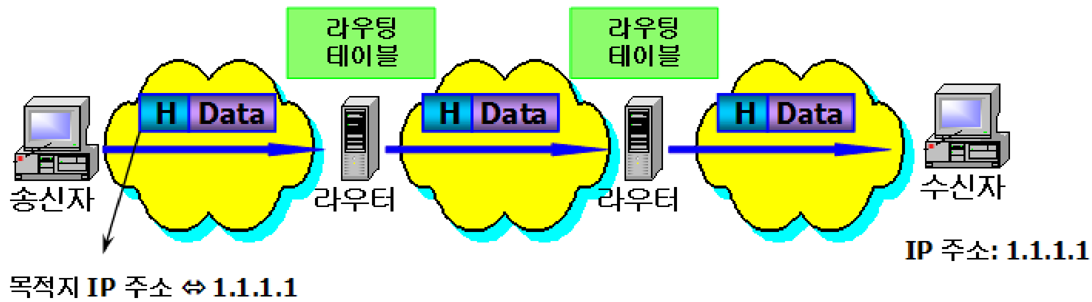


<그림 A.22> 인터넷 패킷 캡슐화 과정

송신 컴퓨터의 패킷은 물리계층에서 비트형태로 인터넷 망으로 전송되며, 인터넷 망에서는 라우팅 과정을 통해 수신 컴퓨터에게 해당 패킷을 전달한다. 수신 컴퓨터에 도착한 패킷은 다시 역캡슐화 과정을 통해 헤더를 제외한 데이터 부분만 상위계층으로 전달된다. 이와 같은 과정을 통해, 최종적으로 송신 사용자가 생성한 데이터가 수신 사용자에게 전달된다.

A.6.4 인터넷 패킷 전달

다음 그림은 IP 프로토콜에 의한 인터넷 패킷 전달과정을 보여준다.



<그림 A.23> 인터넷 패킷 전달

IP 패킷은 헤더(header)와 데이터(data) 부분으로 구성되는데, 헤더 부분에 송신자와 수신자의 IP 주소가 기록된다. IP 패킷전달은 인터넷 상의 라우터(router)에 의해서 이루어진다. 각 라우터는 '라우팅 테이블(routing table)'을 참조하여, 특정 패킷을 어디로 전달(forwarding)해야 할지를 결정한다. 이처럼 여러 라우터의 패킷 전달과정을 통해 최종적으로 IP 패킷이 수신컴퓨터에 도착하게 된다.

라우터들은 인접 라우터와 '라우팅 프로토콜'를 사용하여 라우팅 테이블 정보를 갱신한다. 이와 같은 라우팅 프로토콜에는 RIP(Routing Information Protocol), OSPF(Open Shortest Path First) 및 BGP(Border Gateway Protocol) 등의 프로토콜이 있다.

다음 그림은 "ping6" 프로그램을 사용한 IPv6 패킷 전달과정을 보여준다.

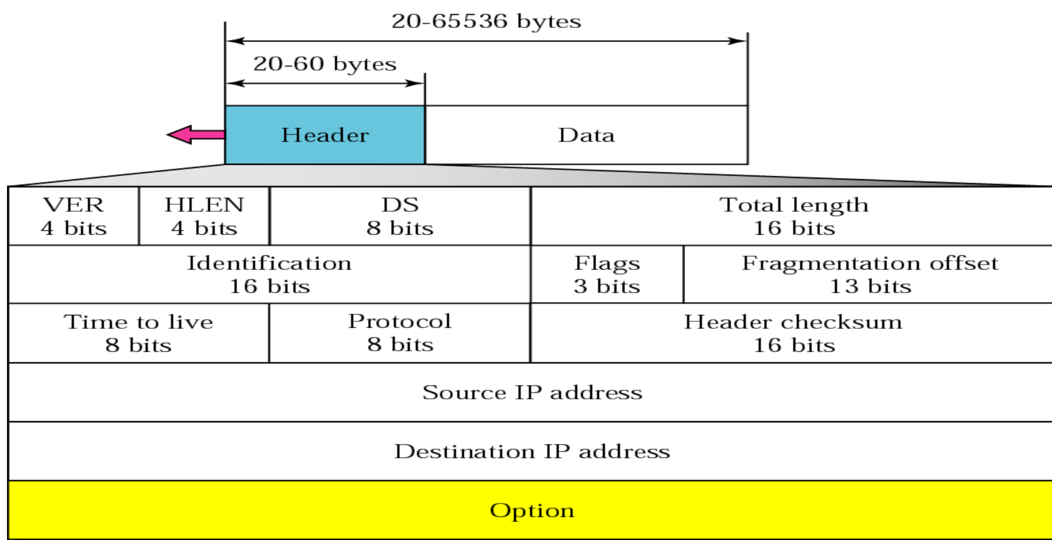
```
C:₩WINDOWS₩system32₩cmd.exe

C:₩Documents and Settings>ping6 www.kame.net

Pinging www.kame.net [2001:200:0:8002:203:47ff:fea5:3085]
from 2002:9be6:6995::9be6:6995 with 32 bytes of data:

Reply from 2001:200:0:8002:203:47ff:fea5:3085: bytes=32 time=274ms
Reply from 2001:200:0:8002:203:47ff:fea5:3085: bytes=32 time=271ms
Reply from 2001:200:0:8002:203:47ff:fea5:3085: bytes=32 time=271ms
Reply from 2001:200:0:8002:203:47ff:fea5:3085: bytes=32 time=270ms

Ping statistics for 2001:200:0:8002:203:47ff:fea5:3085:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 270ms, Maximum = 274ms, Average = 271ms

C:₩Documents and Settings>_
```

<그림 A.24> ping6 사용 예

1.6.5 IP 헤더 포맷

IP 주소는 IP 패킷의 헤더부분에 기록되어 라우터의 패킷 전달과정에 참조된다. 그 밖에 IP 헤더는 여러 가지 다양한 정보를 포함하는데, 웹 프로그래밍의 이해를 위해 헤더 포맷(format)을 알아둘 필요가 있다.

다음 그림은 IPv4 헤더의 구조를 보여준다.



<그림 A.25> IPv4 헤더 구조

IPv4 헤더는 총 20 바이트로 구성되며, 필요에 따라 옵션(option)이 추가될 수 있다. 주요 필드에 대한 설명은 다음과 같다.

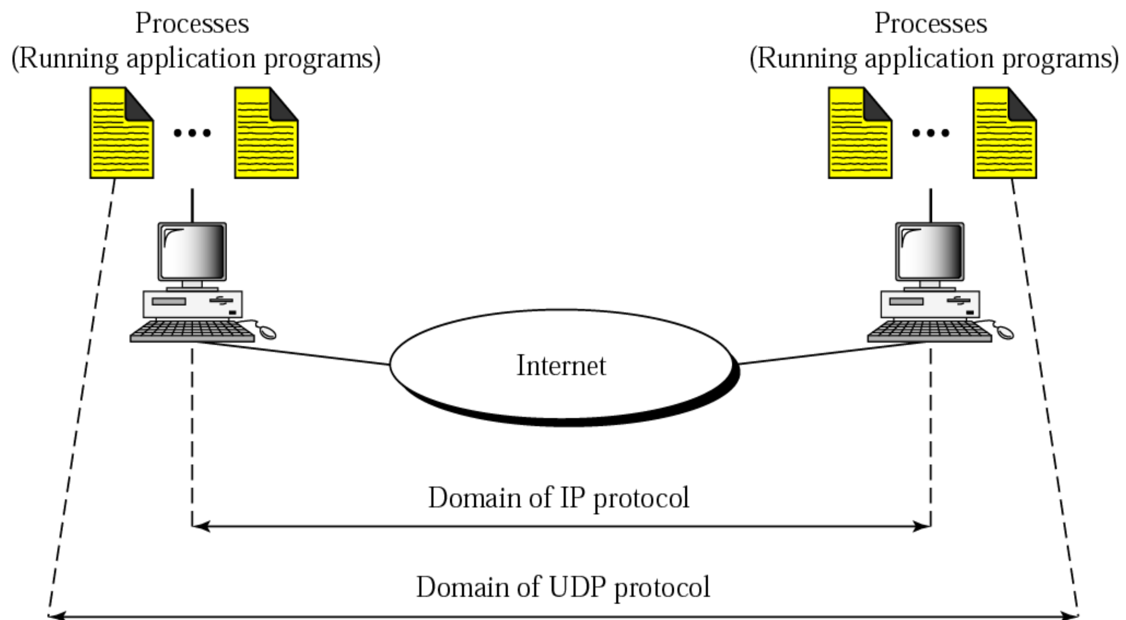
- VER: IPv4 혹은 IPv6 등의 IP 버전을 나타냄
- HLEN: 옵션을 포함한 IPv4 헤더의 길이를 표시함(20~60 바이트; 4 바이트 word 단위)
- DS: Differentiated Service 프로토콜이 사용되는 경우 해당 정보를 표시함
- Total Length: 헤더와 데이터를 포함한 IP 패킷의 전체 길이를 표시함
- Identification, Flags, Fragmentation offsets: IP 패킷이 전송 도중에 fragmentation 되었을 경우에 관련 정보를 표시함
- Time To Live (TTL): IP 패킷이 몇 개의 라우터를 경유하였는지를 표시함

- Protocol: IP 패킷의 데이터가 포함하는 프로토콜 정보를 표시함 (예: TCP, UDP 등)
- Header Checksum: 헤더가 전송 도중에 오류를 경험했는지 검사하기 위해 사용됨
- Source 및 Destination IP 주소: 송수신 컴퓨터의 IP 주소를 기록함

A.7 User Datagram Protocol (UDP)

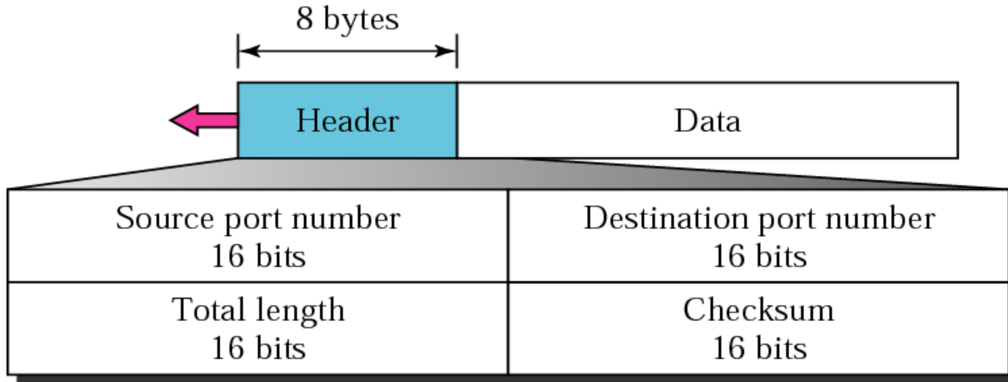
A.7.1 프로세스 통신

IP 프로토콜은 네트워크 계층에서 송신 단말과 라우터, 라우터와 라우터, 라우터와 수신 단말간에 IP 패킷을 전달하기 위해 사용되는 반면에, UDP, TCP, SCTP 프로토콜은 송신 및 수신 단말간에 종단간(end-to-end) 통신을 위해 사용된다. 이를 프로세스간(process-to-process) 통신이라 부른다.



<그림 A.26> 종단 프로세스 통신

UDP 의 주요 기능(사실상 유일한 기능)은 종단 응용 프로그램(혹은 프로세스)간 통신을 위해 포트(port) 번호를 제공하는 것이다. 그림 1-8 처럼 UDP 패킷 헤더는 8 바이트의 고정 길이를 가지며, 2 바이트 크기의 송수신 포트번호를 포함한다.

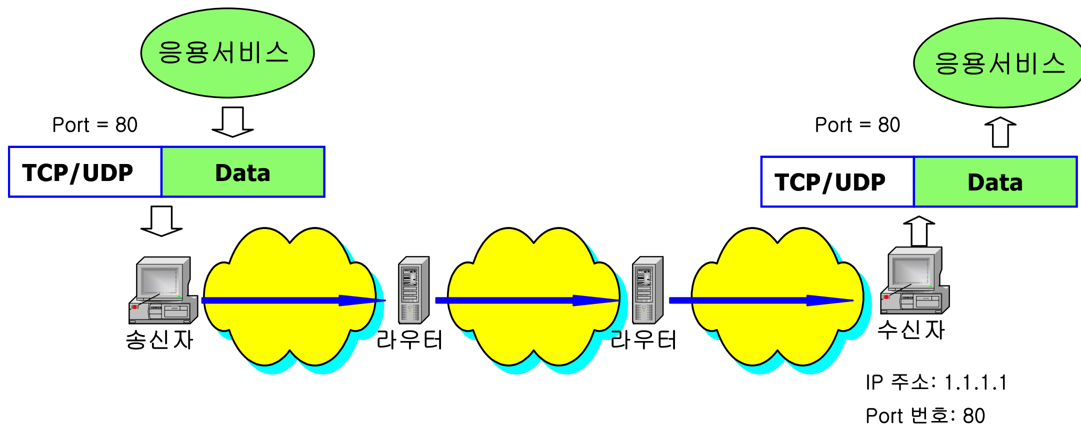


<그림 A.27> UDP 패킷 헤더 구조

A.7.2 포트번호

한편, UDP (TCP, SCTP 포함) 등의 수송계층 패킷의 헤더에는 2 바이트 크기의 포트번호가 포함되어 있으며, 이러한 포트번호는 컴퓨터에서 동작 중인 특정 응용서비스를 식별하는 데에 사용된다. 즉, 컴퓨터에는 웹서비스, 메일서비스 등의 다양한 응용서비스가 동시에 실행될 수 있는데, 그 중에 어떤 응용서비스에 대한 정보를 패킷이 포함하고 있는 지를 나타내기 위해서 포트번호가 TCP, UDP 헤더에 포함되는 것이다.

다음 그림은 포트번호 80 번을 사용하는 웹서비스(WWW)에 대한 패킷전달 과정이다.



<그림 A.28> 포트 번호

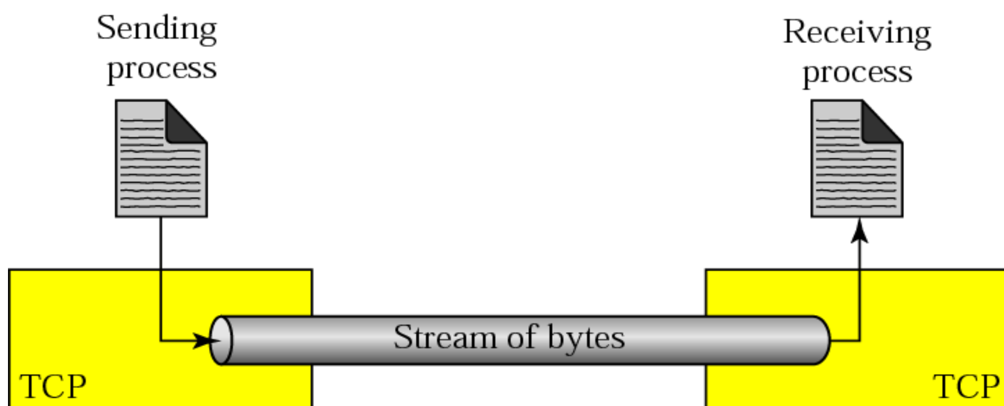
그림에서 송신자는 IP 주소 1.1.1.1 을 가지는 수신자(웹서버)에게 데이터를 전달하기 위해, TCP 헤더의 포트번호를 80 번으로 설정하여 패킷을 전송하고 있다. 해당 패킷이 라우팅 과정을 통해 수신컴퓨터에 도착하였을 때에, 수신컴퓨터는 포트번호가 80 번임을 확인하고 상위 웹 응용 프로그램에 데이터 내용을 전달한다.

포트번호는 이처럼 컴퓨터 도착한 패킷을 어떤 상위 응용 프로그램에 전달할지를 결정하기 위해 사용된다. 예를 들어, 메일서비스는 포트번호 25번을 사용하는 것처럼 응용서비스마다 다른 포트번호를 사용하며, 이러한 포트번호를 "well-known 포트번호"라 한다.

A.8 Transmission Control Protocol (TCP)

TCP는 가장 널리 사용되는 수송계층 프로토콜이다. UDP는 종단 프로세스간 통신을 위해 포트번호 정보만 제공하는 반면에, TCP는 연결(connection)관리, 오류(error) 제어 및 흐름(flow) 제어기능, 혼잡(congestion) 제어 기능 등의 상태(state)관리 기능을 제공한다.

UDP는 '메시지(message)' 단위의 데이터 전송을 수행하는 반면에, TCP는 일련의 바이트(byte) 스트림stream)을 전송한다. 다음 그림은 종단 프로세스간에 TCP를 통해 연속적인 바이트 스트림을 전송하는 기능을 보여준다. 이처럼, 송수신 단말간에 비교적 많은 데이터를 전송하거나 오랜 시간동안 연결 설정이 필요할 때에 TCP를 사용한다.

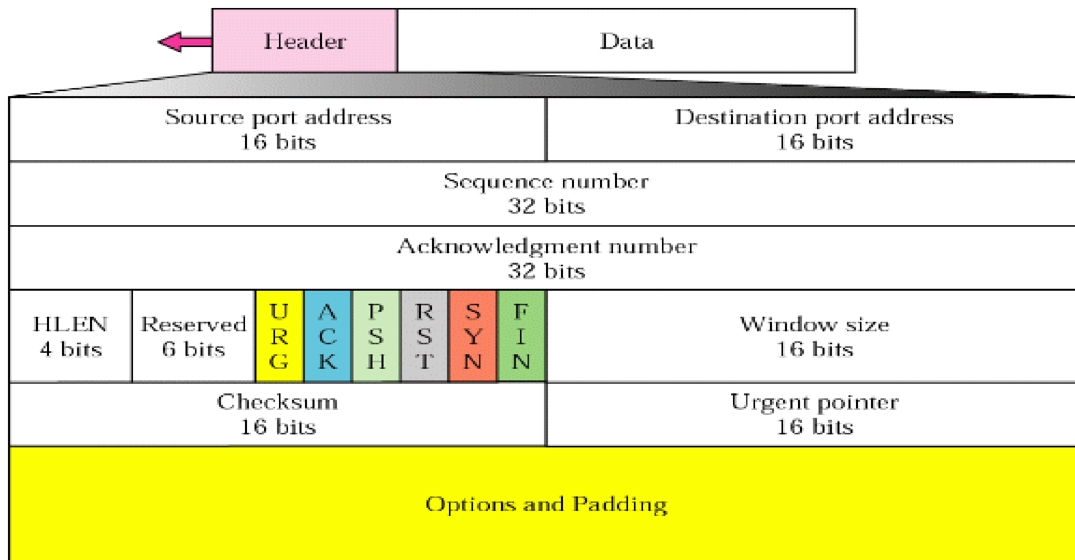


<그림 A.29> TCP 바이트 스트림 전송

위와 같은 바이트 스트림 전송을 위해 TCP는 UDP에 비해 많은 제어 기능을 필요로 한다. 아래에서는 TCP 제어 기능 및 이를 위한 헤더 구조를 살펴보기로 한다.

A.8.1 TCP 패킷 헤더

다음 그림은 TCP 헤더구조를 보여준다. 총 20 바이트의 기본 헤더에 필요에 따라 40 바이트까지 옵션이 추가될 수 있다.



<그림 A.30> TCP 헤더 구조

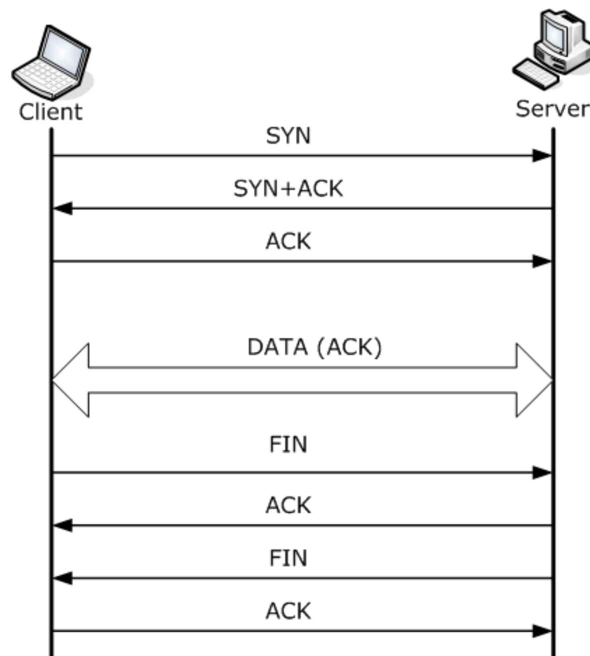
먼저, 첫줄의 4바이트는 UDP처럼 송수신 포트번호 기록을 위해 사용된다. 2번째와 3번째 줄은 TCP 오류, 흐름, 혼잡제어를 위해 사용되는 중요한 정보로서, 바이트 단위의 일련번호(sequence number)를 기록한다. 4번째 및 5번째 줄에는 각종 제어기능에 필요한 정보 및 헤더크기, 그리고 수신 컴퓨터의 버퍼(buffer) 용량을 나타내는 윈도우(window) 정보 등이 있다.

주요 필드(field)에 대한 설명은 다음과 같다.

- 송신(source) 및 수신(destination) 포트번호: 각 2 바이트(16비트)
- Sequence number: 송신자 입장에서, 해당 TCP 패킷의 데이터 부분에 포함된 첫 번째 데이터의 일련 번호(바이트 단위). 즉, 전체 데이터 중에 해당 패킷이 포함하는 데이터 정보를 나타냄
- Acknowledgment number: 수신자 입장에서, 송신자가 보낸 데이터 중에 어느 부분까지 성공적으로 수신하였는지를 관련 일련번호(바이트 단위)로 나타냄. TCP 오류제어에 사용됨
- HLEN: 옵션을 포함한 TCP 헤더의 크기 (20~60바이트; 4바이트 word 단위)

A.8.2 TCP 연결 관리

UDP와 달리 TCP은 연결관리 기능을 제공하며, 이를 통해 데이터 송수신 상태(state)를 지속적으로 관리한다. 아래 그림은 TCP 연결설정 및 연결해제 과정을 보여준다.



<그림 A.31> TCP 연결 관리

TCP 연결 관리는 크게 다음 3가지 과정으로 나뉜다.

1) 연결 설정(connection establishment)

연결을 원하는 client가 먼저 SYN(synchronization) 패킷을 전송하면, server는 이에 대한 ACK(acknowledgment) 정보와 자신의 SYN 정보를 담은 패킷을 전송한다. 이어서 client는 서버의 SYN에 대한 ACK를 전송함으로써 연결 설정을 완료한다. 이처럼 SYN, SYN+ACK, ACK 3개의 패킷을 교환하므로 TCP 연결과정을 3-way handshake 과정이라 부른다.

2) 데이터 송수신(data transfer)

연결설정 후에 client와 server는 데이터 패킷과 상응하는 ACK 패킷을 통해 데이터 전송을 수행

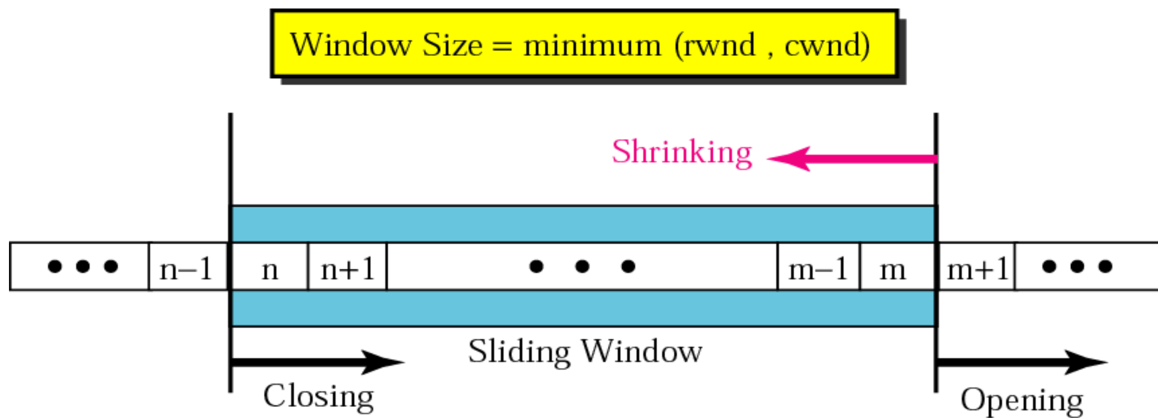
3) 연결 종료(connection shutdown)

연결 종료를 위해 client는 FIN(finish) 패킷을 전송하고 server는 ACK로 응답한다. 이어서, server가 FIN 패킷을 전송하면 client가 ACK로 응답하여 연결을 종료한다. 연결 종료 후에는 더 이상 데이터 전송을 수행할 수 없으며 관련된 모든 상태 정보가 메모리에서 해제된다. 이러한 과정을 4-way handshake 과정이라 부른다.

A.8.3 TCP 오류 및 흐름 제어

TCP는 UDP와 달리 송신한 패킷이 손실(loss)되었을 경우, 재전송(retransmission)을 통해 오류복구기능을 제공한다. 이러한 오류제어 기능은 수신자의 버퍼 용량을 고려하는 흐름(flow) 제어방식과 함께 수행되는 데, 이를 "sliding window" 기반 오류/흐름 제어방식이라 한다.

아래 그림은 sliding window 방식에 따른 송신버퍼에 있는 데이터(패킷)의 흐름을 보여준다.



<그림 A.32> TCP 오류 및 흐름제어를 위한 sliding window

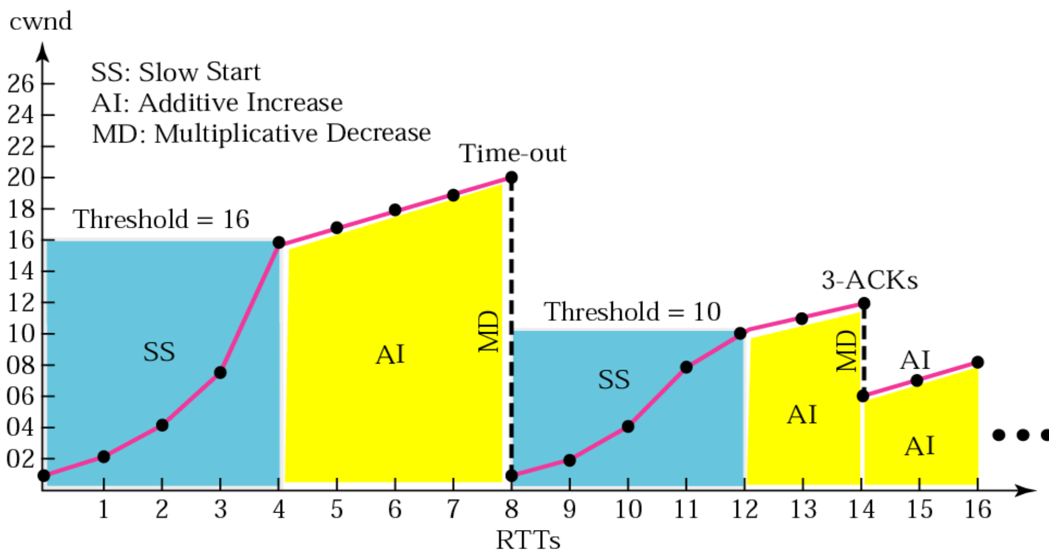
그림에서 n-1 번째 바이트까지의 데이터는 이미 전송이 완료된 상태이고, 송신자는 추가적으로 "n ~ m 번째" 데이터를 보낼 수 있으며, 이를 window 크기라 한다. 이러한 window 크기는 수신자의 버퍼용량(receiver window: rwnd)와 congestion window(cwnd) 크기의 최소값으로 결정된다. cwnd 크기는 다음에 기술하는 혼잡제어 알고리즘에 의하여 결정되는 값이다.

일단 송신된 데이터에 대하여 수신자로부터 해당 ACK 패킷이 (TCP 헤더의 ACK number를 이용) 도착하면, 전송이 완료된 것으로 간주하고 window를 전체적으로 오른쪽으로 이동시킨다(이를 'sliding'이라 함). window 크기에 따라 송신자는 새로운 데이터들을 더 보낼수 있다. 만약 제한된 시간 내에 ACK 패킷이 오지 않는 경우, 송신자는 해당 패킷을 '재전송'함으로써 오류 복구기능을 수행한다. 송신 Window의 크기는 혼잡제어 및 수신버퍼(rwnd) 용량에 따라 축소(shrinking) 될 수도 있다.

A.8.4 TCP 혼잡 제어

혼잡제어는 오류/흐름제어와 함께 TCP의 주요 특징 중의 하나이다. 오류/흐름제어는 종단 송수신자간에 손실 패킷의 재전송 및 수신버퍼 용량을 고려한 안전한 혹은 신뢰적인(reliable) 전송에 중점을 두는 반면에, 혼잡제어(congestion control)에서는 네트워크의 상태에 따라 데이터 송신량을 조절함으로서 전송의 효율성(throughput)을 높이는 방식이다.

구체적으로 TCP 혼잡제어에서는 혼잡윈도우(congestion window: cwnd) 값을 조절하게 되는데, 네트워크가 혼잡상태에 있을 때에서는 cwnd를 낮추고, 정상적인 상태에 있을 때에는 cwnd를 높여주는 알고리즘이다. 다음 그림은 TCP 혼잡제어에 따른 cwnd 변화를 보여준다.



<그림 A.33> TCP 혼잡제어

혼잡제어 과정을 다음 3가지 모드(mode)로 나누어 cwnd 값을 달리 조절한다.

- SS(slow start) 모드: cwnd=1(여기서 1은 패킷 개수를 의미함)로 시작하고, 수신측에서 ACK가 도착하면 2배씩 cwnd를 증가시킨다.

- AI(additive increase) 모드: cwnd값이 threshold 값을 넘게 되면, AI 모드에 들어가며 이때는 ACK 패킷당 '1'씩 cwnd를 증가시킨다.

- MD(multiplicative decrease) 모드: 전송 도중에 손실(loss) 혹은 3개의 중복 ACK가 도착하는 경우, 네트워크 혼잡상태로 판단하고 cwnd 값을 급격히 감소시킨다.

TCP 혼잡제어에서는 이처럼 cwnd 값 조절을 통해, 최대한 안전하게 throughput을 높이고자 하며, 보다 상세한 혼잡제어에 대한 설명은 다른 문헌을 참고하기 바란다.

A.9 Stream Control Transmission Protocol (SCTP)

SCTP는 TCP, UDP에 이어서 3번째로 개발된 수송계층 프로토콜이다. 기본적인 프로토콜 특징은 TCP와 유사하며 다음과 같은 특징을 갖는다.

- TCP 3단계 연결설정 절차 대신에, SCTP는 4 단계 연결설정 절차를 따른다. 즉, 연결설정을 위해 client와 server간에 INIT, INIT-ACK, COOKIE-ECHO, COOKIE-ACK 4개의 패킷교환이 이루어진다. 이는 TCP의 "SYN storm" (정보보안의 DoS 공격에 해당하는 것으로 서버를 다운시키기 위해 공격자가 TCP SYN 패킷을 대량으로 발송하는 행위) 문제 해결을 위해, "cookie" 정보에 대한 확인 절차 위해 4단계가 필요하다.

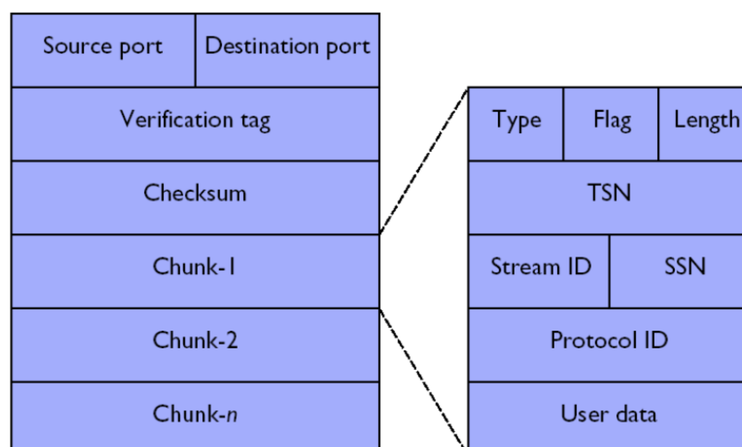
- TCP 4단계 연결종료 절차 대신에, SCTP는 3단계 절차를 수행한다. 즉, SHUTDOWN, SHUTDOWN-ACK, SHUTDOWN-COMPLETE 패킷 교환을 통해 연결을 종료한다.

- TCP는 바이트 기반의 스트림 전송방식인데 비하여, SCTP는 UDP와 유사하게 메시지 기반의 전송 방식을 따른다. 이러한 메시지를 SCTP 청크(chunk)라 부른다. 즉, sequence number 등이 바이트 단위가 아닌 chunk 메시지 단위가 된다.

SCTP는 멀티스트리밍(multi-streaming) 기능과 멀티홈잉(multi-homing) 기능을 제공한다.

A.9.1 SCTP 멀티스트리밍

SCTP는 하나의 연결(association이라 부름)에서 여러 개의 스트림(stream) 데이터를 전송할 수 있다. 즉, 데이터가 여러 종류의 스트림으로 분류될 수 있는 경우에 각 스트림 데이터를 분리하여 전송 할 수 있다. 다음 그림은 SCTP 패킷 구조를 보여준다.



<그림 A.34> SCTP 패킷 구조와 멀티스트리밍

SCTP는 12바이트(포트번호: 4바이트, 검증태그: 4바이트, 체크섬: 4바이트)의 고정 헤더와 다수의 청크(제어 청크와 데이터 청크 포함)로 구성된다. 제어(control) 청크에는 INIT, INIT-ACK, COOKIE-ECHO, COOKIE-ACK 등의 연결설정에 사용되는 청크와, SHUTDOWN, SHUTDOWN-ACK, SHUTDOWN-COMPLETE 등의 연결종료에 사용되는 청크 및 그 외 HEARTBEAT 청크 등의 연결제어에 사용되는 다양한 청크가 있다.

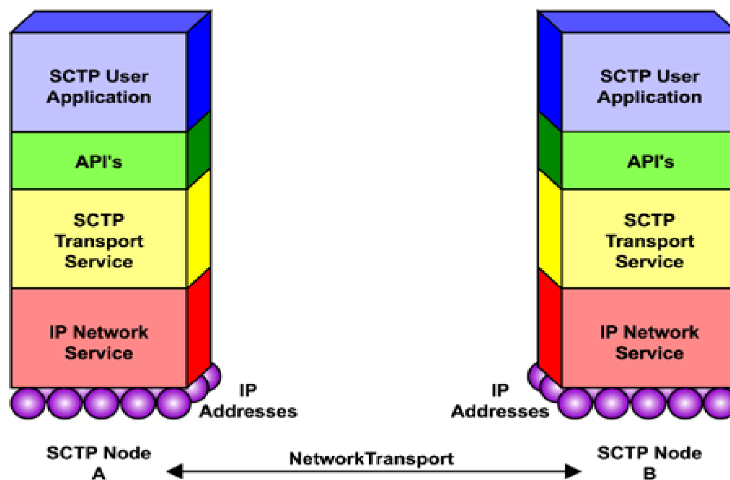
데이터 전송을 위해서는 DATA 청크가 사용되고, DATA 청크는 16바이트의 데이터청크 헤더를 사용하고 다음 필드를 포함한다.

- TSN(Transmission Sequence Number): TCP의 sequence number에 해당되며, SCTP는 DATA 청크별로 TSN 값이 1씩 증가한다. TSN은 오류 및 혼잡제어 등에 사용된다.

- Stream ID 및 SSN(Stream Sequence Number): SCTP의 멀티스트리밍 특성을 지원하기 위한 것으로, 하나의 연결에 여러 스트림 데이터가 각각 별도의 Stream ID를 가질 수 있으며, SSN은 해당 Stream ID에서의 DATA 청크의 일련번호이다. 즉, Stream 별로 고유의 일련번호가 부여되며, 수신측에서 응용 프로세스에 전달된다.

A.9.2 SCTP 멀티홈잉

멀티홈잉(multi-homing)은 SCTP의 독특한 특징 중의 하나로서, 하나의 연결에서 IP 주소를 두 개 이상 사용할 수 있다. 즉, TCP, UDP 연결의 경우 하나의 IP 주소와 하나의 포트로 식별되는 데 비해, SCTP 연결은 하나의 포트와 두 개 이상의 IP 주소를 동시에 사용할 수 있다. 이러한 멀티홈잉 특성을 통해 종단 단말간에 2개 이상의 경로(path)를 사용할 수 있으며, 하나의 경로가 장애(failure)를 일으키는 경우, 곧 바로 다른 경로를 통해 데이터 전송을 유지하게 해주는 기능을 제공한다.

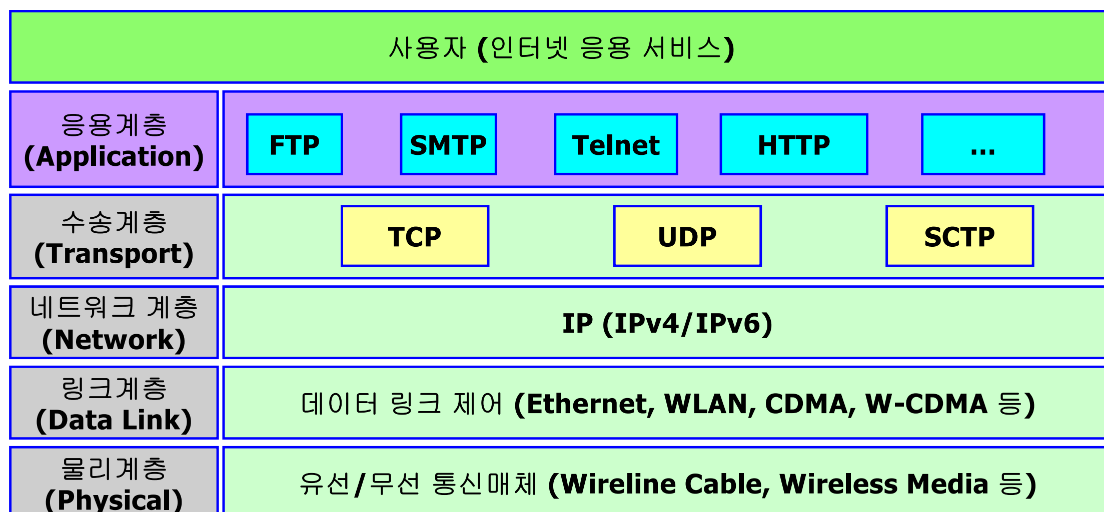


<그림 A.35> SCTP 멀티홈잉

A.10 응용 계층 프로토콜

TCP/IP 패킷은 사용자에게 다양한 인터넷 응용 서비스를 제공하기 위한 데이터 정보를 포함한다. 각 데이터는 상위 응용서비스에 따라 다른 정보 혹은 다른 포맷으로 구성되는데, 이처럼 "응용서비스별로 어떻게 데이터를 구성하고 전송해야 하는지"를 정의한 프로토콜이 응용 계층 프로토콜이다. 즉, 각 응용서비스마다 고유한 응용 프로토콜이 정의된다.

다음 그림은 TCP/IP 프로토콜 스택에서의 응용 계층 프로토콜의 위치이다. 그림 8.23에서처럼, 응용 계층 프로토콜은 상위에 있는 응용서비스 사용자와 하위에 있는 수송계층 프로토콜과의 중간 인터페이스 기능을 제공한다.



<그림 A.36> TCP/IP 응용 프로토콜

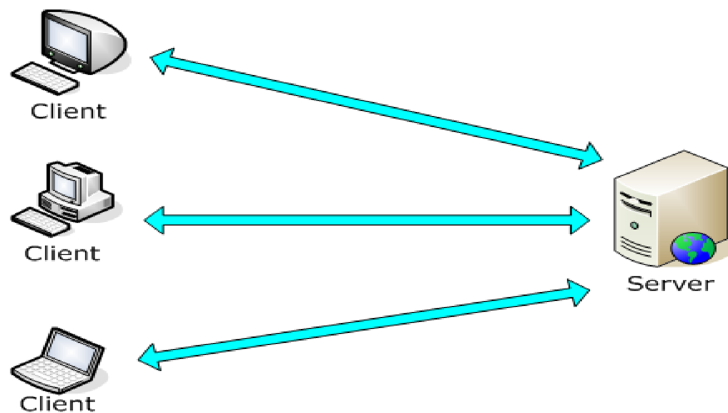
예를 들어, 우리가 지금까지 자주 접해온 인터넷 서비스들은 다음과 같이 상응하는 응용 계층 프로토콜을 기반으로 동작한다.

- 웹서비스(WWW): HTTP (HyperText Transfer Protocol)
- 메일서비스: SMTP (Simple Mail Transfer Protocol)
- 파일전송서비스: FTP (File Transfer Protocol)

A.10.1 클라이언트-서버 모델

우리가 지금까지 접해온 대부분의 인터넷 응용서비스들은 클라이언트-서버(Client-Server) 모델에 따라 동작한다. 예를 들어, WWW 웹서비스를 위해 웹클라이언트와 웹서버가 사용되며, 메일 서비스를 위해 메일 클라이언트와 메일 서버가 동작한다.

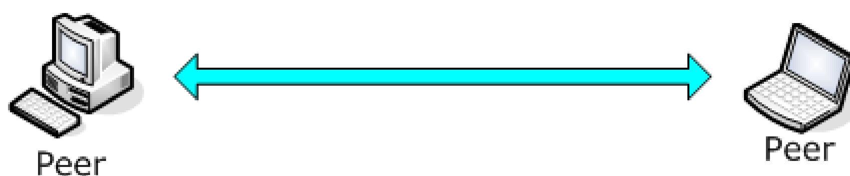
다음 그림은 client-server 모델의 개념을 보여준다.



<그림 A.37> Client-Server 모델

WWW 웹서비스에서 알 수 있듯이, client는 일반적인 서비스 사용자를 의미하고, server는 서비스를 제공하는 사업자를 의미한다. client는 필요할 때에 server에 접속하여(연결하여) 원하는 서비스를 받은 다음에 연결을 종료하는 반면에, server는 1년 365일 동안 불특정 다수의 client 접속을 대기하고 있어야 한다. 대개 server는 다수의 client를 상대하게 된다.

한편, C-S 모델에 대응되는 모델은 Peer-to-Peer (P-P) 모델이다. P-P 모델에서는 두 대의 단말이 동등한 입장에서 통신을 하게 되며, 사실상 각 단말이 client 역할을 하기도 하고 server 역할을 하기도 한다. 대개 먼저 접속을 요청하는 단말이 client 역할을 수행한다. P-P 모델의 전형적인 예제는 VoIP(Voice over IP) 전화 서비스이다.

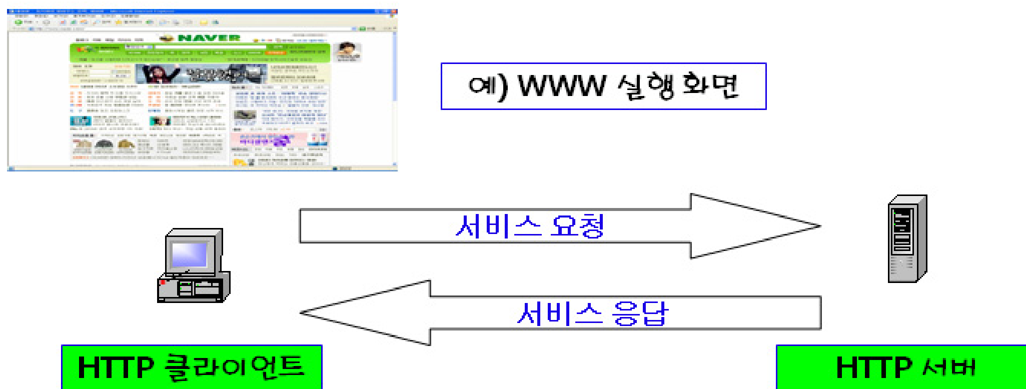


<그림 A.38> Peer-to-Peer 모델

A.10.2 HTTP

먼저 우리가 가장 많이 사용하는 WWW 혹은 웹서비스는 HTTP(HyperText Transfer Protocol)을 사용한다. HTTP 프로토콜은 말 그대로 하이퍼텍스트 정보를 전송하는 프로토콜이다.

다음 그림은 HTTP 프로토콜을 이용한 WWW 문서의 전달과정이다.



<그림 A.39> HTTP 기반 WWW

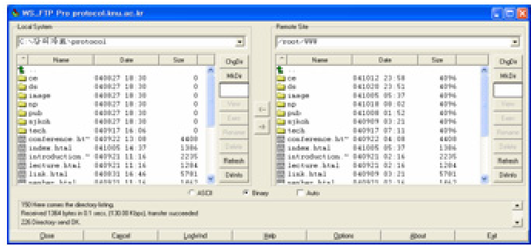
우리가 사용하는 인터넷 익스플로어러에는 HTTP 클라이언트 프로그램이 동작한다. 또한 포털사이트 혹은 웹서버에는 HTTP 서버 프로그램이 동작하고 있다. 우리가 웹브라우저를 사용하여 웹서버로부터 웹문서를 얻기까지의 과정을 정리하면 다음과 같다.

- 웹브라우저는 HTTP 프로토콜을 사용하여 웹 서버에게 서비스(웹문서)를 요청한다. 요청 문서의 위치는 http://www.naver.com/index.html와 같이 URL 로 표시된다.

- 웹서버는 해당 URL 문서를 HTTP 프로토콜을 사용하여 웹브라우저에게 전송한다. HTTP 프로토콜의 데이터 부분에 해당 문서 정도가 포함된다.

A.10.3 FTP

다음 그림은 FTP(File Transfer Protocol)의 클라이언트와 서버간의 동작 모습이다. FTP 클라이언트 프로그램을 사용하여 FTP 서버에 접속한 후 파일을 업로드나 다운로드하는 과정을 볼 수 있다. FTP는 20번 및 21번의 well-known 포트번호를 사용한다.

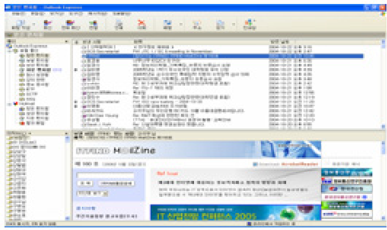


예) ftp 실행 화면

<그림 A.40> FTP

A.10.4 SMTP

다음 그림은 SMTP(Simple Mail Transfer Protocol)의 동작 모습이다. SMTP 프로토콜은 메일 클라이언트가 서버에게 메일 전송을 요청할 때에 사용된다. 반면에 클라이언트가 자신의 메일을 서버로부터 가져오기 위해서는 POP(Post Office Protocol) 혹은 IMAP(Internet Message Access Protocol) 등의 별도 프로토콜을 사용한다



예) Outlook Express 실행 화면

<그림 A.41> SMTP

부록 B. lnp.h 헤더파일

```
/* include lnp.h */
/* Our own header.  Tabs are set for 4 spaces, not 8 */
/* vi:set nonu ts=4 sw=4: */

#ifndef      __lnp_h
#define      __lnp_h

#include     "../config.h"/* configuration options for current OS */
                    /* "../config.h" is generated by configure */

/* If anything changes in the following list of #includes, must change
acsite.m4 also, for configure's tests. */

#include     <sys/types.h>/* basic system data types */
#include     <sys/socket.h>      /* basic socket definitions */
#if TIME_WITH_SYS_TIME
#include     <sys/time.h> /* timeval{} for select() */
#include     <time.h>            /* timespec{} for pselect() */
#else
#if HAVE_SYS_TIME_H
#include     <sys/time.h> /* includes <time.h> unsafely */
#else
#include     <time.h>            /* old system? */
#endif
#endif
#include     <netinet/in.h> /* sockaddr_in{} and other Internet defns */

#ifdef HAVE_NETINET_SCTP_H /* for SCTP Sockets API */
#include     <netinet/sctp.h>
#endif

#include     <arpa/inet.h>/* inet(3) functions */
#include     <errno.h>
#include     <fcntl.h>           /* for nonblocking */
#include     <netdb.h>
#include     <signal.h>
#include     <stdio.h>
#include     <stdlib.h>
#include     <string.h>
#include     <sys/stat.h> /* for S_xxx file mode constants */
#include     <sys/uio.h>         /* for iovec{} and readv/writev */
#include     <unistd.h>
#include     <sys/wait.h>
#include     <sys/un.h>          /* for Unix domain sockets */
```

```
#ifdef HAVE_SYS_SELECT_H
# include    <sys/select.h>      /* for convenience */
#endif

#ifdef HAVE_SYS_SYSCTL_H
#ifdef HAVE_SYS_PARAM_H
# include    <sys/param.h> /* OpenBSD prereq for sysctl.h */
#endif
# include    <sys/sysctl.h>
#endif

#ifdef HAVE_POLL_H
# include    <poll.h>            /* for convenience */
#endif

#ifdef HAVE_SYS_EVENT_H
# include    <sys/event.h> /* for kqueue */
#endif

#ifdef HAVE_STRINGS_H
# include    <strings.h>         /* for convenience */
#endif

/* Three headers are normally needed for socket/file ioctl's:
 * <sys/ioctl.h>, <sys/filio.h>, and <sys/sockio.h>.
 */
#ifdef HAVE_SYS_IOCTL_H
# include    <sys/ioctl.h>
#endif
#ifdef HAVE_SYS_FILIO_H
# include    <sys/filio.h>
#endif
#ifdef HAVE_SYS_SOCKIO_H
# include    <sys/sockio.h>
#endif

#ifdef HAVE_PTHREAD_H
# include    <pthread.h>
#endif

#ifdef HAVE_NET_IF_DL_H
# include    <net/if_dl.h>
#endif
```

```
/* OSF/1 actually disables recv() and send() in <sys/socket.h> */
#ifdef __osf__
#undef recv
#undef send
#define      recv(a,b,c,d) recvfrom(a,b,c,d,0,0)
#define      send(a,b,c,d) sendto(a,b,c,d,0,0)
#endif

#ifndef      INADDR_NONE
/* $$.Ic INADDR_NONE$$ */
#define      INADDR_NONE  0xffffffff   /*    should    have    been    in
<netinet/in.h> */
#endif

#ifndef      SHUT_RD                   /* these three POSIX names are new
*/
#define      SHUT_RD             0      /* shutdown for reading */
#define      SHUT_WR             1      /* shutdown for writing */
#define      SHUT_RDWR    2      /* shutdown for reading and writing */
/* $$.Ic SHUT_RD$$ */
/* $$.Ic SHUT_WR$$ */
/* $$.Ic SHUT_RDWR$$ */
#endif

/* *INDENT-OFF* */
#ifndef INET_ADDRSTRLEN
/* $$.Ic INET_ADDRSTRLEN$$ */
#define      INET_ADDRSTRLEN            16    /* "ddd.ddd.ddd.ddd\0"
                                                  1234567890123456
*/
#endif

/* Define following even if IPv6 not supported, so we can always allocate
   an adequately sized buffer without #ifdefs in the code. */
#ifndef INET6_ADDRSTRLEN
/* $$.Ic INET6_ADDRSTRLEN$$ */
#define      INET6_ADDRSTRLEN    46     /* max size of IPv6 address string:
         "xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx" or
         "xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:ddd.ddd.ddd.ddd\0"
          1234567890123456789012345678901234567890123456 */
#endif
/* *INDENT-ON* */
```

```
/* Define bzero() as a macro if it's not in standard C library. */
#ifndef      HAVE_BZERO
#define      bzero(ptr,n)        memset(ptr, 0, n)
/* $$.If bzero$$ */
/* $$.If memset$$ */
#endif

/* Older resolvers do not have gethostbyname2() */
#ifndef      HAVE_GETHOSTBYNAME2
#define      gethostbyname2(host,family)           gethostbyname((host))
#endif

/* The structure returned by recvfrom_flags() */
struct lnp_in_pktinfo {
  struct in_addr    ipi_addr;    /* dst IPv4 address */
  int                     ipi_ifindex; /* received interface index */
};
/* $$.It lnp_in_pktinfo$$ */
/* $$.Ib ipi_addr$$ */
/* $$.Ib ipi_ifindex$$ */

/* We need the newer CMSG_LEN() and CMSG_SPACE() macros, but few
   implementations support them today.  These two macros really need
    an ALIGN() macro, but each implementation does this differently. */
#ifndef      CMSG_LEN
/* $$.Im CMSG_LEN$$ */
#define      CMSG_LEN(size)            (sizeof(struct cmsghdr) + (size))
#endif
#ifndef      CMSG_SPACE
/* $$.Im CMSG_SPACE$$ */
#define      CMSG_SPACE(size)    (sizeof(struct cmsghdr) + (size))
#endif

/* POSIX requires the SUN_LEN() macro, but not all implementations DefinE
   it (yet).  Note that this 4.4BSD macro works regardless whether there is
   a length field or not. */
#ifndef      SUN_LEN
/* $$.Im SUN_LEN$$ */
# define     SUN_LEN(su) \
       (sizeof(*(su)) - sizeof((su)->sun_path) + strlen((su)->sun_path))
#endif

/* POSIX renames "Unix domain" as "local IPC."
   Not all systems DefinE AF_LOCAL and PF_LOCAL (yet). */
```

```
#ifndef       AF_LOCAL
#define AF_LOCAL    AF_UNIX
#endif
#ifndef       PF_LOCAL
#define PF_LOCAL    PF_UNIX
#endif

/* POSIX requires that an #include of <poll.h> DefinE INFTIM, but many
systems still DefinE it in <sys/stropts.h>.  We don't want to include all
the STREAMS stuff if it's not needed, so we just DefinE INFTIM here. This
is the standard value, but there's no guarantee it is -1. */
#ifndef INFTIM
#define INFTIM        (-1)    /* infinite poll timeout */
/* $$.Ic INFTIM$$ */
#ifdef HAVE_POLL_H
#define       INFTIM_UNPH                       /* tell lnpxti.h we defined
it */
#endif
#endif

/* Following could be derived from SOMAXCONN in <sys/socket.h>, but many
   kernels still #define it as 5, while actually supporting many more */
#define       LISTENQ            1024   /* 2nd argument to listen() */

/* Miscellaneous constants */
#define       MAXLINE            4096   /* max text line length */
#define       BUFFSIZE     8192   /* buffer size for reads and writes */

/* Define some port number that can be used for our examples */
#define       SERV_PORT           9877              /* TCP and UDP */
#define       SERV_PORT_STR "9877"              /* TCP and UDP */
#define       UNIXSTR_PATH "/tmp/unix.str"     /* Unix domain stream */
#define       UNIXDG_PATH         "/tmp/unix.dg"      /*     Unix     domain
datagram */

/* Some things for SCTP */

#define SCTP_PDAPI_INCR_SZ 65535 /* increment size for pdapi when adding
*/
                                                        /* buf space */
#define SCTP_PDAPI_NEED_MORE_THRESHOLD 1024          /*  need  more  space
threshold */
#define SERV_MAX_SCTP_STRM       10     /* normal maximum streams */
#define SERV_MORE_STRMS_SCTP     20     /* larger number of streams */
```

```
/* $$.ix [LISTENQ]~constant,~definition~of$$ */
/* $$.ix [MAXLINE]~constant,~definition~of$$ */
/* $$.ix [BUFFSIZE]~constant,~definition~of$$ */
/* $$.ix [SERV_PORT]~constant,~definition~of$$ */
/* $$.ix [UNIXSTR_PATH]~constant,~definition~of$$ */
/* $$.ix [UNIXDG_PATH]~constant,~definition~of$$ */

/* Following shortens all the typecasts of pointer arguments: */
#define      SA     struct sockaddr

#ifndef HAVE_STRUCT_SOCKADDR_STORAGE
/*
 * RFC 3493: protocol-independent placeholder for socket addresses
 */
#define      __SS_MAXSIZE 128
#define      __SS_ALIGNSIZE      (sizeof(int64_t))
#ifdef HAVE_SOCKADDR_SA_LEN
#define      __SS_PAD1SIZE (__SS_ALIGNSIZE      -      sizeof(u_char)      -
sizeof(sa_family_t))
#else
#define      __SS_PAD1SIZE (__SS_ALIGNSIZE - sizeof(sa_family_t))
#endif
#define      __SS_PAD2SIZE (__SS_MAXSIZE - 2*__SS_ALIGNSIZE)

struct sockaddr_storage {
#ifdef HAVE_SOCKADDR_SA_LEN
       u_char       ss_len;
#endif
       sa_family_t  ss_family;
       char         __ss_pad1[__SS_PAD1SIZE];
       int64_t             __ss_align;
       char         __ss_pad2[__SS_PAD2SIZE];
};
#endif

#define      FILE_MODE     (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
                    /* default file access permissions for new files */
#define      DIR_MODE      (FILE_MODE | S_IXUSR | S_IXGRP | S_IXOTH)
                    /* default permissions for new directories */

typedef      void   Sigfunc(int); /* for signal handlers */

#define      min(a,b)      ((a) < (b) ? (a) : (b))
```

```
#define      max(a,b)      ((a) > (b) ? (a) : (b))

#ifndef      HAVE_ADDRINFO_STRUCT
# include    "../lib/addrinfo.h"
#endif

#ifndef      HAVE_IF_NAMEINDEX_STRUCT
struct if_nameindex {
  unsigned int   if_index;  /* 1, 2, ... */
  char         *if_name;   /* null-terminated name: "le0", ... */
};
/* $$.It if_nameindex$$ */
/* $$.Ib if_index$$ */
/* $$.Ib if_name$$ */
#endif

#ifndef      HAVE_TIMESPEC_STRUCT
struct timespec {
  time_t      tv_sec;             /* seconds */
  long        tv_nsec;      /* and nanoseconds */
};
/* $$.It timespec$$ */
/* $$.Ib tv_sec$$ */
/* $$.Ib tv_nsec$$ */
#endif
/* end lnph */

                    /* prototypes for our own library functions */
int            connect_nonb(int, const SA *, socklen_t, int);
int            connect_timeo(int, const SA *, socklen_t, int);
int     daemon_init(const char *, int);
void    daemon_inetd(const char *, int);
void    dg_cli(FILE *, int, const SA *, socklen_t);
void    dg_echo(int, SA *, socklen_t);
int            family_to_level(int);
char   *gf_time(void);
void    heartbeat_cli(int, int, int);
void    heartbeat_serv(int, int, int);
struct addrinfo *host_serv(const char *, const char *, int, int);
int            inet_srcrt_add(char *);
u_char  *inet_srcrt_init(int);
void    inet_srcrt_print(u_char *, int);
void    inet6_srcrt_print(void *);
char   **my_addrs(int *);
```

```
int            readable_timeo(int, int);
ssize_t        readline(int, void *, size_t);
ssize_t        readn(int, void *, size_t);
ssize_t        read_fd(int, void *, size_t, int *);
ssize_t       recvfrom_flags(int, void *, size_t, int *, SA *, socklen_t *,
struct lnp_in_pktinfo *);
Sigfunc *signal_intr(int, Sigfunc *);
int            sock_bind_wild(int, int);
int            sock_cmp_addr(const SA *, const SA *, socklen_t);
int            sock_cmp_port(const SA *, const SA *, socklen_t);
int            sock_get_port(const SA *, socklen_t);
void    sock_set_addr(SA *, socklen_t, const void *);
void    sock_set_port(SA *, socklen_t, int);
void    sock_set_wild(SA *, socklen_t);
char   *sock_ntop(const SA *, socklen_t);
char   *sock_ntop_host(const SA *, socklen_t);
int            sockfd_to_family(int);
void    str_echo(int);
void    str_cli(FILE *, int);
int            tcp_connect(const char *, const char *);
int            tcp_listen(const char *, const char *, socklen_t *);
void    tv_sub(struct timeval *, struct timeval *);
int            udp_client(const char *, const char *, SA **, socklen_t *);
int            udp_connect(const char *, const char *);
int            udp_server(const char *, const char *, socklen_t *);
int            writable_timeo(int, int);
ssize_t        writen(int, const void *, size_t);
ssize_t        write_fd(int, void *, size_t, int);

#ifdef MCAST
int            mcast_leave(int, const SA *, socklen_t);
int            mcast_join(int, const SA *, socklen_t, const char *, u_int);
int            mcast_leave_source_group(int   sockfd,   const   SA   *src,
socklen_t srclen,  const SA *grp, socklen_t grplen);
int            mcast_join_source_group(int sockfd, const SA *src, socklen_t
srclen,       const SA *grp, socklen_t grplen, const char *ifname, u_int
ifindex);
int            mcast_block_source(int  sockfd,  const  SA  *src,  socklen_t
srclen,       const SA *grp, socklen_t grplen);
int            mcast_unblock_source(int  sockfd,  const  SA  *src,  socklen_t
srclen,  const SA *grp, socklen_t grplen);
int            mcast_get_if(int);
int            mcast_get_loop(int);
int            mcast_get_ttl(int);
```

```
int            mcast_set_if(int, const char *, u_int);
int            mcast_set_loop(int, int);
int            mcast_set_ttl(int, int);

void    Mcast_leave(int, const SA *, socklen_t);
void    Mcast_join(int, const SA *, socklen_t, const char *, u_int);
void    Mcast_leave_source_group(int  sockfd,  const  SA  *src,  socklen_t
srclen,  const SA *grp, socklen_t grplen);
void   Mcast_join_source_group(int sockfd, const SA *src, socklen_t srclen,
const SA *grp, socklen_t grplen, const char *ifname, u_int ifindex);
void    Mcast_block_source(int  sockfd,  const  SA  *src,  socklen_t  srclen,
       const SA *grp, socklen_t grplen);
void    Mcast_unblock_source(int  sockfd,  const  SA  *src,  socklen_t  srclen,
const SA *grp, socklen_t grplen);
int            Mcast_get_if(int);
int            Mcast_get_loop(int);
int            Mcast_get_ttl(int);
void    Mcast_set_if(int, const char *, u_int);
void    Mcast_set_loop(int, int);
void    Mcast_set_ttl(int, int);
#endif

uint16_t     in_cksum(uint16_t *, int);

#ifndef      HAVE_GETADDRINFO_PROTO
int            getaddrinfo(const   char   *,   const   char   *,   const   struct
addrinfo *,
                                   struct addrinfo **);
void    freeaddrinfo(struct addrinfo *);
char   *gai_strerror(int);
#endif

#ifndef      HAVE_GETNAMEINFO_PROTO
int            getnameinfo(const  SA  *,  socklen_t,  char  *,  size_t,  char  *,
size_t, int);
#endif

#ifndef      HAVE_GETHOSTNAME_PROTO
int            gethostname(char *, int);
#endif

#ifndef      HAVE_HSTRERROR_PROTO
const char    *hstrerror(int);
#endif
```

```
#ifndef      HAVE_IF_NAMETOINDEX_PROTO
unsigned int  if_nametoindex(const char *);
char              *if_indextoname(unsigned int, char *);
void               if_freenameindex(struct if_nameindex *);
struct if_nameindex *if_nameindex(void);
#endif

#ifndef      HAVE_INET_PTON_PROTO
int                inet_pton(int, const char *, void *);
const char    *inet_ntop(int, const void *, char *, size_t);
#endif

#ifndef      HAVE_INET_ATON_PROTO
int           inet_aton(const char *, struct in_addr *);
#endif

#ifndef      HAVE_PSELECT_PROTO
int           pselect(int, fd_set *, fd_set *, fd_set *,
                         const struct timespec *, const sigset_t *);
#endif

#ifndef      HAVE_SOCKATMARK_PROTO
int           sockatmark(int);
#endif

#ifndef      HAVE_SNPRINTF_PROTO
int           snprintf(char *, size_t, const char *, ...);
#endif

                    /* prototypes for our own library wrapper functions */
void    Connect_timeo(int, const SA *, socklen_t, int);
int           Family_to_level(int);
struct addrinfo *Host_serv(const char *, const char *, int, int);
const char          *Inet_ntop(int, const void *, char *, size_t);
void                 Inet_pton(int, const char *, void *);
char                *If_indextoname(unsigned int, char *);
unsigned int         If_nametoindex(const char *);
struct if_nameindex *If_nameindex(void);
char   **My_addrs(int *);
ssize_t       Read_fd(int, void *, size_t, int *);
int           Readable_timeo(int, int);
ssize_t      Recvfrom_flags(int, void *, size_t, int *, SA *, socklen_t *,
struct lnp_in_pktinfo *);
```

```
Sigfunc *Signal(int, Sigfunc *);
Sigfunc *Signal_intr(int, Sigfunc *);
int             Sock_bind_wild(int, int);
char    *Sock_ntop(const SA *, socklen_t);
char    *Sock_ntop_host(const SA *, socklen_t);
int             Sockfd_to_family(int);
int             Tcp_connect(const char *, const char *);
int             Tcp_listen(const char *, const char *, socklen_t *);
int             Udp_client(const char *, const char *, SA **, socklen_t *);
int             Udp_connect(const char *, const char *);
int             Udp_server(const char *, const char *, socklen_t *);
ssize_t         Write_fd(int, void *, size_t, int);
int             Writable_timeo(int, int);

        /* prototypes for our Unix wrapper functions: see {Sec errors} */
void    *Calloc(size_t, size_t);
void     Close(int);
void     Dup2(int, int);
int             Fcntl(int, int, int);
void     Gettimeofday(struct timeval *, void *);
int             Ioctl(int, int, void *);
pid_t    Fork(void);
void    *Malloc(size_t);
int      Mkstemp(char *);
void    *Mmap(void *, size_t, int, int, int, off_t);
int             Open(const char *, int, mode_t);
void     Pipe(int *fds);
ssize_t         Read(int, void *, size_t);
void     Sigaddset(sigset_t *, int);
void     Sigdelset(sigset_t *, int);
void     Sigemptyset(sigset_t *);
void     Sigfillset(sigset_t *);
int             Sigismember(const sigset_t *, int);
void     Sigpending(sigset_t *);
void     Sigprocmask(int, const sigset_t *, sigset_t *);
char    *Strdup(const char *);
long     Sysconf(int);
void     Sysctl(int *, u_int, void *, size_t *, void *, size_t);
void     Unlink(const char *);
pid_t    Wait(int *);
pid_t    Waitpid(pid_t, int *, int);
void     Write(int, void *, size_t);

/* prototypes for our stdio wrapper functions: see {Sec errors} */
```

```
void    Fclose(FILE *);
FILE   *Fdopen(int, const char *);
char   *Fgets(char *, int, FILE *);
FILE   *Fopen(const char *, const char *);
void    Fputs(const char *, FILE *);

/* prototypes for our socket wrapper functions: see {Sec errors} */
int             Accept(int, SA *, socklen_t *);
void    Bind(int, const SA *, socklen_t);
void    Connect(int, const SA *, socklen_t);
void    Getpeername(int, SA *, socklen_t *);
void    Getsockname(int, SA *, socklen_t *);
void    Getsockopt(int, int, int, void *, socklen_t *);
#ifdef HAVE_INET6_RTH_INIT
int             Inet6_rth_space(int, int);
void   *Inet6_rth_init(void *, socklen_t, int, int);
void    Inet6_rth_add(void *, const struct in6_addr *);
void    Inet6_rth_reverse(const void *, void *);
int             Inet6_rth_segments(const void *);
struct in6_addr *Inet6_rth_getaddr(const void *, int);
#endif
#ifdef HAVE_KQUEUE
int             Kqueue(void);
int             Kevent(int, const struct kevent *, int,
                            struct kevent *, int, const struct timespec *);
#endif
void    Listen(int, int);
#ifdef HAVE_POLL
int             Poll(struct pollfd *, unsigned long, int);
#endif
ssize_t         Readline(int, void *, size_t);
ssize_t         Readn(int, void *, size_t);
ssize_t         Recv(int, void *, size_t, int);
ssize_t         Recvfrom(int, void *, size_t, int, SA *, socklen_t *);
ssize_t         Recvmsg(int, struct msghdr *, int);
int             Select(int, fd_set *, fd_set *, fd_set *, struct timeval *);
void    Send(int, const void *, size_t, int);
void    Sendto(int, const void *, size_t, int, const SA *, socklen_t);
void    Sendmsg(int, const struct msghdr *, int);
void    Setsockopt(int, int, int, const void *, socklen_t);
void    Shutdown(int, int);
int             Sockatmark(int);
int             Socket(int, int, int);
void    Socketpair(int, int, int, int *);
```

```
void   Writen(int, void *, size_t);

int           Sctp_recvmsg(int s, void *msg, size_t len,
                            struct sockaddr *from, socklen_t *fromlen,
                            struct     sctp_sndrcvinfo     *sinfo,     int
*msg_flags);
int          Sctp_sendmsg (int s, void *data, size_t len,
                                  struct sockaddr *to, socklen_t tolen,
                                  uint32_t ppid, uint32_t flags, uint16_t
stream_no,
                                  uint32_t timetolive, uint32_t context);
int   Sctp_bindx(int sock_fd,struct sockaddr_storage *at,int num, int op);

void   err_dump(const char *, ...);
void   err_msg(const char *, ...);
void   err_quit(const char *, ...);
void   err_ret(const char *, ...);
void   err_sys(const char *, ...);

#endif /* __lnp_h */
```

“이 교재는 2008년도 삼성전자 정보통신트랙과
경북대학교 초일류 모바일-디스플레이산업
인력양성사업단에 의하여 지원되었음.”