# BeeJ's Guide to Network Programming.

# 인터넷 소켓 활용(v.1.5.4, 17-May-1998)

http://www.ecst.csuchico.edu/~beej/guide/net 번역 : 박성호(tempter@fourthline.com),1998/8/20

---

## 시작

소켓 프로그램이 어렵나요? 그냥 맨페이지만 보고서는 알아내기가 좀 어렵나요? 뭔가 있어보이는 인터넷 프로그램을 만들고 싶지만 `bind()`를 호출하고 `connect()`를 호출하고 이런 저런 구조체를 뒤지고 할 시간이 없나요?

글쎄요, 제가 그 지겨운걸 다 해놓았고요, 여러분과 이 정보를 공유하고 싶군요. 바로 찾아오셨습니다. 이 문서가 바로 평균적인 C 프로그래머에게 네트워크 프로그램에 관련된 정보를 드릴겁니다.

---

## 대상

이 문서는 안내서이지 리퍼런스는 아닙니다. 아마도 소켓 프로그래밍을 처음 시작하면서 어디서부터 해야 할지 모르는 사람들에게 도움이 될겁니다. 물론 어떤 의미에서도 이 글은 소켓 프로그래밍에 관한 완벽한 안내서는 아닐 겁니다. 단지 도저히 의미를 알 수 없던 맨페이지들을 조금씩 이해하게 되기만 바랄 뿐입니다.

---

## 사용도구

대부분의 코드는 리눅스 PC에서 GNU의 gcc를 이용하여 컴파일 되었습니다. 또한 HPUX에서 gcc를 이용해서 컴파일 된다는 것도 확인했습니다. 그러나 모든 작은 코드들이 테스트 된것은 아니라는 것을 기억하시기 바랍니다.

(이하 존칭 생략)

---

## 내용


- 소켓이란 무엇인가.
- 두가지 종류의 소켓
- 네트워크 이론과 저수준의 알수없는 것들
- `struct`s--이걸 모르면 외계인이 지구를 파괴할걸~~
- 순서 바꾸기
- IP주소 와 활용법
- `socket()`--파일 기술자를 잡아라
- `bind()`--나는 어떤 포트에 연결되었나?
- `connect()`--어이 거기!
- `listen()`--누가 전화좀 걸어주지
- `accept()`--포트3490에 전화걸어주셔서 감사합니다.
- `send()` and `recv()`--말좀 해봐!
- `sendto()` and `recvfrom()`--말좀해봐! 데이터그램방식
- `close()` and `shutdown()`--꺼지쇼!
- `getpeername()`--누구십니까?
- `gethostname()`--난 누구인가?
- DNS--"whitehouse.gov", - "198.137.240.100"
- 클라이언트 서버의 배경
- 간단한 스트림서버

- 간단한 스트림클라이언트
- 데이터그램 소켓
- 블로킹
- select()--동기화된 중복입출력, 대단하군!
- 참고사항
- 주의사항 및 연락처


---

# 소켓이란 무엇인가.

소켓이란 단어는 많이 들었을 것이다. 그리고 아마도 그 소켓이 정확히 무엇인가에 대하여 궁금해 하기도 했을 것이다. 소켓은 정규 유닉스 파일 기술자를 이용하여 다른 프로그램과 정보를 교환하는 방법을 의미한다.

뭐라고라?

좋다. 아마도 유닉스를 잘하는 사람들이 이렇게 얘기하는 것을 들어본 적이 있을 것이다. "유닉스에서는 모든게 파일로 되어있군!" 실제로 그들이 얘기하는 것은 모든 유닉스 프로그램들이 어떤 종류의 입출력을 하더라도 파일 기술자를 통해서 하게 된다는 것이다. 파일 기술자는 사실 열려진 파일을 의미하는 정수일 뿐이다. 그러나 그 파일은 네트워크가 될수도 있고 FIFO, 파이프, 터미널, 실제 디스크상의 파일이 될수도 있으며 그 밖의 무엇도 다 된다는 것이다. 유닉스의 모든것은 파일이다! 따라서 당신이 인터넷을 통하여 멀리 떨어진 다른 프로그램과 정보를 교환하기 위해서는 파일 기술자를 이용하면 된다는 것이다. 믿으쇼~

"똑똑이 양반, 그 파일 기술자는 도대체 어떻게 만드는거요?" 라는게 당신의 맘속에 지금 막 떠오른 질문일 것이다. 여기에 대답이 있다. socket()을 호출하면 소켓 기술자를 얻게 되고 send(), recv()등의 소켓에 관련된 함수를 호출하여 정보를 교환할 수 있다. (man send, man recv를 해봐도 됨)

"잠깐!" 이렇게 이의를 제기하겠지. "그 소켓 기술자가 파일 기술자라면 도대체 왜 read(),write()를 쓰면 안되는거요?" 짧게 말하면 맞다. 그러나 send(),recv()를 쓰는 것이 여러모로 네트워크를 통한 정보전달을 제어하기에 도움이 된다는 것이다.

다음은 뭔가? 소켓의 종류는? DARPA 인터넷 주소(인터넷 소켓), 경로명과 지역노드(유닉스 소켓), CCITT X.25 주소(X.25 소켓, 그냥 무시해도 됨)등이 있고 아마도 당신이 쓰는 유닉스에 따라서 더 많은 종류의 소켓들이 있을 것이다. 이 문서는 첫번째 (인터넷 소켓) 하나만 설명할 것이다.

---

# 두가지 종류의 소켓

인터넷 소켓에 두가지 종류가 있나? 그렇다. 음..사실은 거짓말이다. 좀 더있긴 하지만 겁을 주고 싶지 않기 때문에 이것 두가지만 이야기 하는 것이다. RAW 소켓이라는 매우 강력한 것도 있으며 한번 봐두는 것도 좋다.

두가지 종류는 무엇인가? 하나는 스트림소켓 이고 다른 하나는 데이터그램 소켓이다. 이후에는 SOCK_STREAM, SOCK_DGRAM으로 지칭될 것이다. 데이터그램 소켓은 비연결 소켓이라고도 한다. (비록 그 소켓에서도 원한다면 connect()를 사용할 수도 있다. connect()절을 참조할것)

스트림 소켓은 양측을 신뢰성있게 연결해 주는 소켓이다. 만약 두가지 아이템을 이 소켓을 통하여 보낸다면 그 순서는 정확히 유지될 것이다. 에러까지 교정된다. 만일 에러가 생긴다면 당신 실수이고 당신실수를 막는 방법은 여기서 설명하지 않을 것이다.

스트림 소켓은 어디에 쓰이는가? 아마도 텔넷이라고 들어봤을 것이다. 들어봤느뇨? 그게 이 소켓을 쓴다. 입력한 모든 글자는 그 순서대로 전달이 되야 하는 경우이다. 사실 WWW사이트의 포트 80에 텔넷으로 접속하여 "GET pagename"을 입력하면 HTML 화일의 내용이 우르르 나올 것이다.

어떻게 스트림 소켓이 이정도의 정확한 전송 품질을 갖추게 되는가? 이 소켓은 TCP를 이용하기 때문이다. (Transmission Control Protocol, RFC-793에 무척 자세하게 나와있다.) 아마도 TCP 보다는 TCP/IP를 더 많이 들어봤을 것이다. 앞부분은 바로 이 TCP이고 뒷부분의 IP는 인터넷 라우팅을 담당하는 프로토콜이다.

괜찮군~ 데이터그램 소켓은 어떤가? 왜 비연결이라고 하는지? 내용에 무슨 관련이 있는지? 왜 신뢰도가 떨어지지? 사실 이 소켓의 경우 당신이 데이터그램을 보낸다면 정확히 도착할 수도 있다. 또는 패킷들의 순서가 바뀌어서 도착할 수도 있다. 그러나 만약 도착한다면 그 내용은 사실 정확한 것이다.

데이터그램 소켓 또한 라우팅에는 IP를 이용하지만 TCP는 이용하지 않는다. 사실은 UDP(RFC-768)을 이용한다.

연결을 안하는가? 스트림 소켓에서처럼 열려있는 연결을 관리할 필요가 없는 것이다. 그냥 데이터 패킷을 만들어서 목적지에 관련된 IP헤더를 붙여서 발송하기만 하면 되는 것이다. 연결이 필요없다. 보통 tftp나 bootp 에 사용되는 것이다.

좋아! 그러면 데이터 패킷이 도착하지 않을지도 모르는 이런 걸 어떻게 실제 프로그램에서 사용하지? 사실 프로그램들은 UDP위에 그 나름대로의 대책을 갖추고 있는 것이다. 예를 들면 tftp같은 경우에는 하나의 패킷을 보낸 후에 상대편이 잘 받았다는 응답 패킷이 올때까지 기다리는 것이다. 만약 일정시간(예를 들면 5초)동안 응답이 없으면 못받은 것으로 간주하고 다시 보내고, 다시 보내고 응답이 있으면 다음 패킷을 보내고 하게 되는것이다. 이 잘받았다는 응답(ACK reply) 방식은 사실 SOCK_DGRAM을 사용할 경우 매우 중요하다.

---

# 네트워크 이론과 저아래의 알수없는 것들

간단히 프로토콜의 레이어에 대해서 언급을 했지만(UDP위에 나름대로의 대책 어쩌구) 이제는 실제로 네트워크가 어떻게 작동하는 지를 알아볼 때가 되었고 실제로 SOCK_DGRAM이 어떻게 구성되는 지를 알아볼 필요가 있을 것같다. 사실 이 절은 그냥 넘어가도 된다.

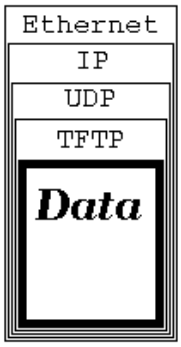

**Figure 1. Data Encapsulation**

여러분~ 이제는 데이타 캡슐화에 대하여 배우겠어요~ 사실 이것은 매우 중요하다. 얼마나 중요하냐면 우리 학교에서 네트워크 코스를 통과하려면 반드시 알아야 하는 사항이기 때문이다. (흠..) 내용은 이렇다. 데이터 패킷이 만들어지면 먼저 첫번째 프로토콜(tftp 프로토콜)에 필요한 머리말과 꼬리말이 붙는다. 이렇게 한번 캡슐화된 내용은 다시 두번째 프로토콜(UDP)에 관련된 머리말과 꼬리말이 다시 붙게 된다. 그 다음에는 IP, 그 다음에는 마지막으로 하드웨어 적인 계층으로서 이더넷 프로토콜로 캡슐화가 되는 것이다.

다른 컴퓨터에서 이 패킷을 받게 되면 하드웨어가 이더넷 헤더를 풀고 커널에서 IP와 UDP 헤더를 풀고 tftp 프로그램에서 tftp헤더를 풀고 하여 끝으로 원래의 데이터를 얻게 되는 것이다.

이제 드디어 악명높은 계층적 네트워크 모델(Layered Network Model)을 얘기할 때가 된것 같다. 이 모델은 다른 모델들에 비해서 네트워크의 시스템을 기술하는 측면에서 많은 이점이 있다. 예를 들면 소켓 프로그래밍을 하는 경우 더 낮은 계층에서 어떤 물리적인 방식(시리얼인지 thin ethernet인지 또는 AUI방식인지)으로 전달되는 지에 대하여 전혀 신경을 쓰지 않고도 작업이 가능해 질 수 있다는 것이다. 실제 네트워크 장비나 토폴로지는 소켓 프로그래머에게는 전혀 관계없는 분야이다.

더이상 떠들지 않고 다음 계층들을 일러 주는데 만일 네트워크 코스에서 시험을 보게 될 경우라면 외우는 것이 좋을 것이다.

- Application
- Presentation
- Session
- Transport
- Network
- Data Link
- Physical

물리적 계층(Physical layer)는 하드웨어(시리얼, 이더넷등) 이다. 어플리케이션 계층은 상상할 수 있듯이 물리적 계층의 반대편 끝이다. 이 계층을 통하여 사용자는 네트워크와 접촉하게 되는 것이다.

사실 이 모델은 자동차 수리 설명서 처럼 실질적인 뭔가를 할 수 있기에는 너무나 일반적인 얘기이다. 유닉스의 경우를 들어 보다 실질적인 얘기를 해 본다면,

- Application Layer (*telnet, ftp, etc.*)
- Host-to-Host Transport Layer (*TCP, UDP*)
- Internet Layer (*IP and routing*)
- Network Access Layer (*was Network, Data Link, and Physical*)

이러한 계층으로 살펴 본다면 아까의 데이터 캡슐화가 각각 어떤 계층에 속하는 가를 알 수 있을 것이다.

이렇게 많은 작업이 하나의 데이터 패킷을 만드는데 동원되는 것이다. 이 내용을 당신이 데이터의 패킷 머리부분에 몽땅 타이핑 해 넣어야 한다는 얘기다. (물론 농담이다.) 스트림 소켓의 경우 데이터를 내보내기 위해 해야 할 일은 오직 `send()`를 호출하는 것 뿐이다. 데이터 그램의 경우에는 원하는 방식으로 데이터를 한번 캡슐화하고 (tftp방식등) `sendto()`로 보내버리면 되는 것이다.커널이 전송계층과 인터넷 계층에 관련된 캡슐화를 하고 나머지는 하드웨어가 한다. 아~ 첨단 기술!!

이것으로 간단한 네트워크 이론은 끝이다. 참, 라우팅에 관해서 하고 싶던 얘기들을 하나도 안했다. 흠, 하나도 없다. 정말이지 라우팅에 관해서 하나도 얘기하지 않을 것이다. 라우터가 IP헤더를 벗겨내서 라우팅 테이블을 참조하여 어쩌구 저쩌구...만일 정말로 여기에 관심이 있다면 IP RFC를 참조할 것이며 만약 거기에 대해서 하나도 알지 못한다면! 생명에 지장은 없다.

---

## struct S

결국은 여기까지 왔군. 드디어 프로그래밍에 관한 얘기를 할 때이다. 이 절에서는 실제로 꽤나 이해하기 어려운 소켓 인터페이스에서 쓰이는 여러가지 데이터 타입에 대한 얘기를 할 예정이다.

먼저 쉬운것. 소켓 기술자이다.소켓 기술자의 데이터 형은

```
int
```

이다. 그냥 보통 int이다. (정수형)

뭔가 좀 이상하더라도 그냥 참고 읽기 바란다. 이것은 알아야 한다. 정수에는 두 바이트가 있는데 상위 바이트가 앞에 있거나 또는 하위 바이트가 앞에 있게 된다. 앞의 경우가 네트워크 바이트 순서이다. 어떤 호스트는 내부적으로 네트워크 바이트 순서로 정수를 저장하는 경우도 있으나 안그런 경우가 많다. 만일 NBO라고 언급된 정수가 있다면 함수를 이용하여 (`htons()`함수) 호스트 바이트 순서로 바꾸어야 한다. 만약 그런 언급이 없다면 그냥 내버려 둬도 된다.

첫번째 구조체, `struct sockaddr`. 이 구조체는 여러가지 형태의 소켓 주소를 담게된다.

```
struct sockaddr {
    unsigned short    sa_family;    /* address family, AF_xxx       */
    char              sa_data[14];  /* 14 bytes of protocol address */
};
```

`sa_family` 는 여러가지가 될 수 있는데, 이 문서에서는 그중에서 "`AF_INET`"인 경우만 다루게 된다. `sa_data` 는 목적지의 주소와 포트번호를 가지게 된다. 약간 비실용적이군.

`sockaddr` 구조체를 다루기 위해서는 다음과 같은 `parallel structure`를 만들어야 한다. ("in"은 인터넷을 의미한다.)

```
struct sockaddr_in {
    short int          sin_family;  /* Address family               */
    unsigned short int sin_port;    /* Port number                  */
    struct in_addr     sin_addr;    /* Internet address             */
    unsigned char      sin_zero[8]; /* Same size as struct sockaddr */
};
```

이 구조체는 각각의 항을 참조하기가 좀더 쉬운 것 같다. 주의할 점은 `sin_zero`배열은 `sockaddr` 과 구조체의 크기를 맞추기 위해서 넣어진 것이므로 `bzero()`나 `memset()`함수를 이용하여 모두 0으로 채워져야 한다. 또한 꽤 중요한 점인데, 이 구조체는 `sockaddr` 의 포인터를 이용하여 참조될 수 있고 그 반대도 가능하다는 것이다. 따라서 `socket()`함수가 `struct sockaddr *` 를 원하더라도 `struct sockaddr_in`을 사용할 수 있고 바로 참조할 수도 있는 것이다. 또한 `sin_family`는 `sa_family`에 대응되는 것이며 물론 "`AF_INET`"로 지정되어야 하며 `sin_port, sin_addr`은 네트워크 바이트 순서로 되어야 하는 점이 중요한 것이다.

그러나! 어떻게 struct in_addr sin_addr 전체가 NBO가 될 수 있는가? 이 질문은 살아남은 가장 뭣같은 유니온인 struct in_addr 에 대한 보다 신중한 검토가 필요할 것같다.

```
/* Internet address (a structure for historical reasons) */
struct in_addr {
    unsigned long s_addr;
};
```

음.. 이것은 유니온 "이었었"다. 그러나 그런 시절은 지나갔다. 시원하게 없어졌군! 따라서 만약 "ina"를 struct sockaddr_in형으로 정의해 놓았다면 ina.sin_addr.s_addr 로 NBO 상태의 4바이트 인터넷 어드레스를 정확하게 참조할 수 있을 것이다. 만약 사용하는 시스템이 struct in_addr에 그 끔찍한 유니온을 아직도 사용하고 있더라도 #defines S 덕분에 위에 한것과 마찬가지로 정확하게 참조할 수는 있을 것이다.

---

# 순서 바꾸기

이제 다음 절로 왔다. 네트워크와 호스트 바이트 순서에 대해서 말이 너무 많았고 이제는 실제 움직일 때라고 본다.

좋다. 두가지 형태의 변환이 있는데 하나는 short(2 바이트)와 long(4바이트)의 경우이다. 이 함수들은 unsigned변수에서도 잘 작동된다. 이제 short변수를 호스트 바이트 순서에서 네트워크 바이트 순서로 변환하는 경우를 보자. 호스트의 h로 시작해서 to 를 넣고 네트워크의 n 을 넣은 후 short의 s 를 넣는다. 그래서 htons()이다. (읽기는 호스트 투 네트워크 쇼트이다.)

너무 쉬운가?

사실 h,n,s,l 의 어떤 조합도 사용가능하다. (물론 너무 바보스러운 조합을 하지는 않겠지..예를 들어 stolh, 쇼트 투 롱 호스트?? 이런건 없다. 적어도 이 동네에서는없다.) 있는 것들은 다음과 같다.

- htons()--"Host to Network Short"
- htonl()--"Host to Network Long"
- ntohs()--"Network to Host Short"
- ntohl()--"Network to Host Long"

아마도 이제 상당히 많이 알게된 것같이 생각들을 할 것이다. "char의 바이트 순서를 어떻게 바꾸지?(역자주: 이 질문은 아마 의미없는 질문으로 한 것 같은데 답도 없고 더이상의 언급이 없는 것으로 보아 빼고 싶은 부분이다.)" 또는 "염려마, 내가 쓰는 68000 기계는 이미 네트워크 바이트 순서로 정수를 저장하니까 변환할 필요는 없어 " 라고 생각할 수도 있을 것이다. 그러나 꼭 그렇지만은 않다. 그렇게 작성된 프로그램을 다른 기계에서 작동시킨다면 당연히 문제가 발생할 것이다. 여기는 유닉스 세계고 이기종간의 호환성은 매우 중요한 것이다. 반드시 네트워크에 데이터를 보내기 전에 네트워크 바이트 순서로 바꿔서 보낸다는 것을 기억할 지어다.

끝으로 sin_addr, sin_port는 네트워크 바이트 순서로 기록하는데 왜 sin_family는 안 그러는가? 답은 간단하다. sin_addr과 sin_port는 캡슐화되어 네트워크로 전송되어야 하는 변수인 것이다. 따라서 당연히 NBO여야 한다. 그러나 sin_family는 시스템 내부에서 커널에 의해서만 사용되는 변수이며 네트워크로 전송되지 않는 것이므로 호스트 바이트 순서로 기록되어야 하는 것이다.

---

# IP주소는 무엇이며 어떻게 다루는가?

다행스럽게도 IP주소를 산정해 주는 수많은 함수들이 있으며 따라서 4바이트의 long변수에 직접 계산해서 << 연산자를 이용해서 집어넣어야 하는 수고는 할 필요가 없다.

먼저 struct sockaddr_IN ina가 정의되어 있고 132.241.5.10 이 IP 주소이며 이 값을 변수에 넣어야 한다고 가정해 보자. inet_addr()함수가 바로 이럴때 사용하는 것이다. 그 함수는 숫자와 점으로 구성된 IP주소를 unsigned long 변수에 집어 넣어 준다. 다음과 같이 하면 된다.

ina.sin_addr.s_addr = inet_addr("132.241.5.10")

inet_addr()는 결과값으로 이미 NBO인 값을 돌려주며 굳이 htonl()을 또 사용할 필요는 없다는 점에 주의해야 한다. 멋지군!

그러나 위의 짤막한 코드는 그렇게 견실해 보이진 않는다. 왜냐하면 inet_addr()은 에러의 경우 -1을 돌려주게 되며 unsigned long에서 -1은 255.255.255.255를 의미한다. 이는 인터넷 브로드캐스트 어드레스가 된다. 나쁜 녀석. 항상 에러 처리를 확실히 하는것이 좋다.

좋다. 이제 IP주소를 long에 넣는것은 알았는데 그 반대는 어떻게 할 것인가? 만약에 값이 들어있는 struct in_addr은 가지고 있는데 이를 숫자와 점으로 표시하려면? 이 경우는 inet_ntoa()를 쓰면 된다.(ntoa 는 네트워크 투 아스키이다.)

```
printf("%s",inet_ntoa(ina.sin_addr));
```

위의 코드는 IP주소를 프린트 해 줄것이다. 이 함수는 long 변수가 아니라 struct in_addr 를 변수로 받아 들인다는 점을 주의해야 한다. 또한 이 함수는 char 에 대한 포인터를 결과로 돌려 주는데 이는 함수내에 static 한 공간에 저장되며 따라서 매번 함수가 호출될 때마다 이 포인터가 가리키는 곳의 값은 변화한다는 것이다. 즉 예를 들면,

```
char *a1, *a2;
.
.
a1 = inet_ntoa(ina1.sin_addr);  /* this is 198.92.129.1 */
a2 = inet_ntoa(ina2.sin_addr);  /* this is 132.241.5.10 */
printf("address 1: %sWn",a1);
printf("address 2: %sWn",a2);
```

의 출력은 이렇게 나올 것이다.

```
address 1: 132.241.5.10
address 2: 132.241.5.10
```

만약에 이 값을 저장해야 할 필요가 있다면 strcpy()를 이용하여 고유의 char 배열에 저장해야 할 것이다.

이절에서 얘기할 것은 다 했다. 나중에 "whitehouse.gov" 문자열을 해당하는 IP주소로 바꾸는 법을 알려 줄것이다. (DNS절 참조)

---

# socket() ; 파일 기술자를 잡아라

안하면 맞을것 같아서 socket() 시스템 호출에 대해서 얘기해야만 할것같다. 이걸 잠깐 보자.

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

int socket(int domain, int type, int protocol);
```

그런데 이 변수들은 또 뭔가? 첫째 domain 은 struct sockaddr_in 에서처럼 AF_INET 로 지정하면 된다. 다음 type 은 SOCK_STREAM이나 SOCK_DGRAM으로 지정하면 된다. 끝으로 protocol은 0으로 지정하면 된다. (언급하지 않았지만 더 많은 domain과 더 많은 type 이 있다는 것을 기억하라. socket() 맨페이지를 참고하고 또한 protocol 에 대해서 좀더 알려면 getprotobyname()을 참조하면 된다.)

socket()은 바로 나중에 사용할 소켓 기술자인 정수값을 돌려주며 에러시에는 -1을 돌려주게 된다. 전역변수인 errno에 에러값이 기록된다. (perror()의 맨페이지를 참조할것.)

---

# bind() ; 나는 어떤 포트에 연결되었나?

일단 소켓을 열게 되면 이 소켓을 현재 시스템의 포트에 연결시켜 주어야 한다. (이 작업은 보통 listen()함수를 이용해서 외부의 접속을 대기할 때 시행되며 일반적으로 머드게임 사이트들이 telnet *.*.*.* 6969 로 접속하라고 할때도 이 작업을 시행했다는 의미이다. ) 만약에 그저 다른 호스트에 연결하기만 할 예정이라면 그냥 connect()를 사용하여 연결만 하면 되고 이 작업은 필요가 없다.

아래는 bind() 시스템 호출의 선언이다.

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

int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

sockfd는 socket()함수에서 얻은 소켓 기술자이며 my_addr은 IP 주소에 관한 정보(즉, IP 주소와 포트번호)를 담고 있는 struct sockaddr 에 대한 포인터 이고 addrlen은 그 구조체의 사이즈(sizeof(struct sockaddr))이다.

휴~~ 한방에 받아들이기에는 좀 그렇군. 예를 보자.

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

#define MYPORT 3490

main()
{
    int sockfd;
    struct sockaddr_in my_addr;

    sockfd = socket(AF_INET, SOCK_STREAM, 0); /* do some error checking! */

    my_addr.sin_family = AF_INET;     /* host byte order */
    my_addr.sin_port = htons(MYPORT); /* short, network byte order */
    my_addr.sin_addr.s_addr = inet_addr("132.241.5.10");
    bzero(&(my_addr.sin_zero), 8);    /* zero the rest of the struct */

    /* don't forget your error checking for bind(): */
    bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));
    .
    .
    .
```

몇가지 주의할 점은 my_addr.sin_port 는 my_addr.sin_addr.s_addr과 같이 NBO이다. 또한 헤더화일은 각각의 시스템마다 다를 수 있으므로 각자의 시스템의 맨 페이지를 참고해야 할 것이다.

마지막으로 bind()와 관련해서 주소나 포트의 지정이 때에 따라서 자동화 될 수도 있다는 것을 언급해야 할 것같다.

```
    my_addr.sin_port = 0; /* choose an unused port at random */
    my_addr.sin_addr.s_addr = INADDR_ANY;  /* use my IP address */
```

my_addr.sin_port를 0으로 지정하면 자동으로 사용되지 않고 있는 포트 번호를 지정해 줄것이며 my_addr.sin_addr.s_addr를 INADDR_ANY로 지정할 경우 현재 작동되고 있는 자신의 IP주소를 자동으로 지정해 주게 된다.

만약 여기서 약간만 주의를 기울였다면 INADDR_ANY를 지정할 때 NBO로 바꾸는 것을 빼먹은 것을 눈치챌 것이다. 나아쁜~~. 그러나 난 내부정보를 알고 있지롱. 사실은 INADDR_ANY는 0이다. 0은 순서를 바꾸어도 0인것이다. 그러나 순수이론적인 측면에서 INADDR_ANY가 그러니까 12정도인 세계가 존재한다면 이 코드는 작동 안할것이다. 그래서? 난 상관없다. 정 그렇다면,

```
    my_addr.sin_port = htons(0); /* choose an unused port at random */
    my_addr.sin_addr.s_addr = htonl(INADDR_ANY);  /* use my IP address */
```

이제는 믿기 어려울 정도로 이식가능한 코드가 되었다. 다만 지적하고 싶은 것은 작동하는 데에는 아무 문제가 없다는 점이다.

bind()또한 에러가 났을때 -1을 돌려주며 errno에 에러의 코드가 남게 된다.

bind()를 호출할 때 주의할점 : 절대 제한선 아래로 포트번호를 내리지 말라는 것이다. 1024 아래의 번호는 모두 예약되어 있다. 그 위로는 65535까지 원하는 대로 쓸 수가 있다. (다른 프로그램이 쓰고 있지 않은 경우에 한해서..)

또 하나의 작은 꼬리말 : bind() 를 호출하지 않아도 되는 경우가 있다. 만일 다른 호스트에 연결 (connect())하고자 하는 경우에는 자신의 포트에는 (텔넷의 경우처럼)전혀 신경 쓸 필요가 없다. 단지 connect()를 호출하기만 하면 알아서

bind가 되어 있는지를 체크해서 비어있는 포트에 bind를 해준다.

---

## connect() ; 어이~ 거기~

이제 잠깐만 마치 자신이 텔넷 프로그램인 것처럼 생각해 보기로 하자. 당신의 사용자는 명령하기를 (TRON영화에서처럼.. (역자: 난 그 영화 안 봤는데..)) 소켓 기술자를 얻어오라 했고 당신은 즉시 socket()를 호출했다. 다음에 사용자는 132.241.5.10 에 포트 23(정규 텔넷 포트번호)에 연결하라고 한다. 윽, 이젠 어떻게 하지?

다행스럽게도 당신(프로그램)은 connect()절(어떻게 연결하는가)를 심각하게 읽고 있으며 당신의 주인을 실망시키지 않으려고 미친듯이 읽어나가는 중이로다~~

connet()는 다음과 같이 선언한다.

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

int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

sockfd는 이제는 친숙해진 소켓 기술자이며 serv_addr은 연결하고자 하는 목적지인 서버의 주소와 포트에 관한 정보를 담고 있는 struct sockaddr 이며 addrlen은 앞에서 이야기 한것과 같이 그 구조체의 크기이다.

뭔가 좀 이해가 갈듯 하지 않은가? 예를 들어 보자.

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

#define DEST_IP   "132.241.5.10"
#define DEST_PORT 23

main()
{
    int sockfd;
    struct sockaddr_in dest_addr;   /* will hold the destination addr */

    sockfd = socket(AF_INET, SOCK_STREAM, 0); /* do some error checking! */

    dest_addr.sin_family = AF_INET;        /* host byte order */
    dest_addr.sin_port = htons(DEST_PORT); /* short, network byte order */
    dest_addr.sin_addr.s_addr = inet_addr(DEST_IP);
    bzero(&(dest_addr.sin_zero), 8);       /* zero the rest of the struct */

    /* don't forget to error check the connect()! */
    connect(sockfd, (struct sockaddr *)&dest_addr, sizeof(struct sockaddr));
    .
    .
    .
```

다시 말하건데 connect()의 결과값을 한번 체크해 봐야 한다. 에러일 경우 -1을 돌려주고 errno를 세팅하기 때문이다.

또한 bind()를 호출하지 않은 것에 주의해야 한다. 기본적으로 여기서는 자신의 포트 번호에는 결코 관심이 없기 때문이다. 단지 어디로 가는가만이 중요하다. 커널이 알아서 로컬 포트를 선정해 줄 것이며 우리가 연결하고자 하는 곳에서는 자동으로 이 정보를 알게 될 것이다.

---

## listen() ; 누가 전화좀 걸어주지~

이제 보조를 바꾸어서, 만약에 어디론가 연결하고자 하는 것이 아니라 외부로부터의 접속을 대기해서 접속이 올 경우 어떤 방식으로든지 간에 처리를 해 주어야 하는 경우라면 어찌 할 것인가. 이 작업은 두 단계로 이루어진다. 먼저 listen()을 해야 되고 그 다음에 accept()를 해야 된다는 것이다.

listen()은 상당히 간단하지만 약간의 설명은 필요하다.

```
int listen(int sockfd, int backlog);
```

sockfd는 보통의 소켓 기술자이며 backlog는 접속대기 큐의 최대 연결 가능 숫자이다. 그건 또 뭔 얘기인가? 외부로부터의 연결은 이 대기 큐에서 accept()가 호출될 때까지 기다려야 한다는 것이며 숫자는 바로 얼마나 많은 접속이 이 큐에 쌓여질 수 있는가 하는 것이다. 대부분의 시스템은 이 숫자를 조용하게 20정도에서 제한하고 있으며 보통은 5에서 10 사이로 지정하게 된다.

또 다시 listen()도 에러의 경우 -1을 돌려주며 errno를 세팅한다.

아마 상상할 수 있듯이 listen()보다 앞서서 bind()를 호출해야 하며 만약에 bind()가 되지 않으면 우리는 랜덤하게 지정된 포트에서 외부의 접속을 기다려야 한다. (포트를 모르고서 누가 접속할 수 있겠는가? 우엑~~) 따라서 외부의 접속을 기다리는 경우라면 다음 순서대로 작업이 진행되어야 하는 것이다.

```
socket();
bind();
listen();
/* accept() goes here */
```

위의 것만으로도 이해가 갈만하다고 보고 예제에 대신하겠다. (accept()절에 보다 괜찮은 코드가 준비되어 있다.) 이 모든 sha-bang(역자: 이 뭐꼬?)중에서 가장 헷갈리는 부분은 accept()를 부르는 부분이다.

---

## accept() ; 포트 3490에 전화걸어주셔서 감사합니다.

준비! accept()를 호출하는 것은 뭔가 좀 수상하긴 하다. 과연 뭐가 벌어지는가? 저 멀리 떨어진 곳에서 누군가가 connect()를 호출하여 당신이 listen()을 호출하고 기다리는 포트에 접속을 시도한다. 그들의 연결은 바로 accept()가 호출되기 까지 큐에서 바로 당신이 accept()를 호출하여 그 연결을 지속하라고 명령할 때까지 대기하게 된다. 그러면 이 함수는 오로지 이 연결을 위한 완전히 신제품 소켓 파일 기술자를 돌려주게 된다. 갑자기 당신은 하나값으로 두개의 소켓 기술자를 갖게 되는 것이다. 원래의 것은 아직도 그 포트에서 연결을 listen()하고 있다. 또 하나는 새롭게 창조되어 드디어 send()와 recv()를 할 준비가 되도록 하는 것이다.

드디어 여기까지 왔다! 감격~~

선언은 아래와 같다.

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

int accept(int sockfd, void *addr, int *addrlen);
```

sockfd는 listen()하고 있는 소켓의 기술자이다. 뻔하지 뭐.. addr은 로컬 struct sockaddr_in의 포인터이다. 여기에 들어온 접속에 관한 정보가 담겨지게 되고 이를 이용해서 어느 호스트에서 어느 포트를 이용해서 접속이 들어왔는지를 알 수 있게 된다. addrlen은 로컬 정수 변수이며 이 정수에는 struct sockaddr_in의 크기가 미리 지정되어 있어야 한다. 이 숫자보다 더 많은 바이트의 정보가 들어오면 accept()는 받아 들이지 않을 것이며 적데 들어온다면 addrlen의 값을 줄여 줄 것이다.

accept() 는 에러가 났을 경우에 어떻게 한다고? -1을 돌려주고 errno 를 세팅한다.

아까 맨치로 한방에 받아들이기에는 좀 그러니까 예제를 열심히 읽어 보자.

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

#define MYPORT 3490    /* the port users will be connecting to */

#define BACKLOG 10     /* how many pending connections queue will hold */

main()
{
    int sockfd, new_fd;  /* listen on sock_fd, new connection on new_fd */
    struct sockaddr_in my_addr;    /* my address information */
    struct sockaddr_in their_addr; /* connector's address information */
    int sin_size;
```

```
    sockfd = socket(AF_INET, SOCK_STREAM, 0); /* do some error checking! */

    my_addr.sin_family = AF_INET;         /* host byte order */
    my_addr.sin_port = htons(MYPORT);     /* short, network byte order */
    my_addr.sin_addr.s_addr = INADDR_ANY; /* auto-fill with my IP */
    bzero(&(my_addr.sin_zero), 8);        /* zero the rest of the struct */

    /* don't forget your error checking for these calls: */
    bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));

    listen(sockfd, BACKLOG);

    sin_size = sizeof(struct sockaddr_in);
    new_fd = accept(sockfd, &their_addr, &sin_size);
    .
    .
    .
```

이제 new_fd를 이용해서 send()와 recv()를 이용할 수 있다는 것이다. 만약 원한다면 더이상의 연결을 받아들이지 않고 하나의 연결만 이용하기 위해서 close()를 이용하여 원래의 sockfd를 막아 버릴 수도 있다.

---

## send(), recv() ; 말좀해봐~

이 두 함수는 스트림 소켓이나 연결된 데이터그램 소켓위에서 정보를 주고 받을때 사용하는 것들이다. 만약 보통의 비연결 데이터그램 소켓을 사용한다면 sendto()와 recvfrom()절을 참조하도록 한다.

send() 호출의 선언은 아래와 같다.

```
int send(int sockfd, const void *msg, int len, int flags);
```

sockfd는 socket()를 통해서 얻었거나 accept()를 통해서 새로 구한, 데이터를 보낼 소켓의 기술자이며, msg는 보낼 데이터를 가리키는 포인터, len은 보낼 데이터의 바이트 수 이며 flags 는 그냥 0으로 해야 한다. (플래그에 관한 보다 자세한 내용은 send()의 맨 페이지를 참조할것.)

약간의 예제가 다음과 같다.

```
char *msg = "Beej was here!";
int len, bytes_sent;
.
.
len = strlen(msg);
bytes_sent = send(sockfd, msg, len, 0);
.
.
.
```

send()는 결과값으로 보내진 모든 바이트 수를 돌려주는데 이것은 보내라고 한 숫자보다 작을 수도 있다. 가끔은 보내고자 하는 데이터의 크기가 미처 감당하지 못할 만한 숫자인 경우도 있으며 이 경우 send()는 자기가 감당할 수 있는 숫자만큼만 보내고 나머지는 잘라 버린후 당신이 그 나머지를 다시 보내 줄 것으로 기대하는 것이다. 만약에 보내라고 한 데이터의 크기보다 작은 숫자가 결과값으로 돌아 왔다면 그 나머지 데이터를 보내는 것은 전적으로 당신의 책임인 것이다. 그나마 희소식은 데이터의 사이즈가 작다면 (1k 이내라면) 아마도 한번에 모두 보낼 수 있을 것이다. 또한 에러의 경우 -1을 돌려주며 errno를 세팅한다.

recv()의 경우도 상당히 유사하다.

```
int recv(int sockfd, void *buf, int len, unsigned int flags);
```

sockfd는 읽어올 소켓의 기술자이며 buf는 정보를 담을 버퍼이다. len은 버퍼의 최대 크기이고 flags는 0으로 세팅해야 한다. (자세한 flags의 정보는 recv() 맨 페이지를 참조할것.)

recv()는 실제 읽어들인 바이트 숫자를 돌려주며 에러의 경우는 -1, errno를 세팅한다.

쉬웠을까? 쉬웠지.. 이제 당신은 스트림 소켓을 이용해서 데이터를 보내고 받을 수 있게 되었다. 우와~ 유닉스 네트워크 프로그래머네~~

---

## sendto(), recvfrom() ; 말좀해봐~ 데이터 그램 방식

괜찮은걸, 이라고 말하고 있는줄로 생각하겠다. 그런데 데이터그램에 관한 나머지는 어딨지? 노프라블레모~ 아미고~ (역자: 터미네이터2가 생각나는군~~) 이제 할 것이다.

데이터그램 소켓은 연결을 할 필요가 없다면 데이터를 보내기 전에 주어야 할 나머지 정보는 어떻게 주어야 하는가? 맞다. 목적지의 주소를 알려주어야 한다. 여기에 예제가 있다.

```
int sendto(int sockfd, const void *msg, int len, unsigned int flags,
           const struct sockaddr *to, int tolen);
```

보다시피 이 함수는 두가지 부가정보가 더 들어간 것 이외에는 기본적으로 send()와 동일하다. to 는 struct sockaddr의 포인터이며(아마도 struct sockaddr_in) 여기에는 목적지의 주소와 포트번호가 담겨 있어야 할 것이다. tolen은 그 구조체의 크기인 것이다.

send()와 마찬가지로 sendto()도 보내어진 데이터의 바이트수를 결과로 돌려주며(실제 보내라고 준 데이터의 크기보다 작을지도 모르는), 에러의 경우 -1을 돌려준다.

비슷하게 recvfrom()도 아래와 같다.

```
int recvfrom(int sockfd, void *buf, int len, unsigned int flags
             struct sockaddr *from, int *fromlen);
```

역시 이것도 두가지 변수가 더 주어지게 된다. from은 데이터를 보내는 장비의 주소와 포트를 담고 있는 struct sockaddr 이며 fromlen은 로컬 정수변수로서 구조체의 크기가 세팅되어 있어야 한다. 함수가 호출된 뒤에는 fromlen에는 실제 from의 크기가 수록되게 된다.

recvfrom()은 실제 받은 데이터의 바이트수를 돌려주며 에러의 경우는 -1, errno를 세팅하게 된다.

만약 connect()를 이용하여 데이터그램 소켓을 연결한 후의 상황이라면 간단히 send(), recv() 를 사용해도 상관 없으며 소켓 인터페이스는 자동으로 목적지와 소스에 관한 정보를 함수에 추가해서 작동되게 될 것이다.

---

## close(), shutdown() ; 꺼지쇼.

휴~~ 하루종일 데이터를 보내고 받았더니..이제는 소켓을 닫을 때가 된 것이다. 이건 쉽다. 정규 파일 기술자에 관한 close()를 사용하면 되는 것이다.

```
close(sockfd);
```

이것으로 더이상의 입출력은 불가능 해지며 누구든지 원격지에서 이 소켓에 읽고 쓰려고 하는 자는 에러를 받게 될 것이다.

약간 더 세밀한 제어를 위해서는 shutdown()을 사용하면 된다. 이것을 이용하면 특정방향으로의 통신만을 끊을 수도 있게 된다.

```
int shutdown(int sockfd, int how);
```

sockfd는 소켓 기술자이며 how는 다음과 같다.

- 0 - 더이상의 수신 금지
- 1 - 더이상의 송신 금지
- 2 - 더이상의 송수신 금지(close()와 같은 경우)

shutdown() 은 에러의 경우 -1을 돌려주며 errno를 세팅한다.

황송하옵게도 연결도 되지않은 데이터그램 소켓에 shutdown()을 사용한다면 단지 send(), recv()를 사용하지 못하게만 만들 것이다. connect()를 사용한 경우에만 이렇게 사용할 수 있다는 것을 기억해야 한다. (역자: 그렇다면 sendto, recvfrom은 사용이 된다는 얘기인가??테스트가 필요할듯.)

암것도 아니군.

---

## getpeername() ; 누구십니까?

이 함수는 되게 쉽다.

너무 쉬워서 절을 따로 만들 필요가 없지않나 고민했지만 여기 있는걸 보니까..

getpeername()은 상대편 쪽 스트림 소켓에 누가 연결되어 있는가를 알려준다.

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

int getpeername(int sockfd, struct sockaddr *addr, int *addrlen);
```

sockfd는 연결된 스트림 소켓의 기술자이며 addr은 상대편의 정보를 담게 될 struct sockaddr(또는 struct sockaddr_in)의 포인터 이며 addrlen은 정수를 가리키는 포인터로서 구조체의 크기가 지정되어 있어야 한다.

에러의 경우는 -1을 돌려주고 errno를 세팅한다. (외우겠군.)

일단 주소를 알게되면 inet_ntoa()나 gethostbyaddr()을 이용하여 좀더 많은 정보를 알아낼 수 있게 되지만 상대편의 login name을 알게되는 것은 아니다. (만일 상대편에 ident 데몬이 돌고 있다면 알아낼 방법이 없는 것은 아니지만 이 내용은 이 글의 취지를 벗어나는 내용이므로 RFC-1413을 참조하라고 말하고 싶다.)

---

## gethostname() ; 난 누구인가?

getpeername()보다 더 쉬운 것이 이 함수이다. 결과로 프로그램이 돌고 있는 컴퓨터의 이름을 알려준다. 이름은 gethostbyname()을 이용하여 로컬 장비의 IP주소를 알아내는데 사용될 수도 있다.

뭐가 더 재미있는가? 몇가지 생각해 볼 수 있는데 이 문서에는 적절하지 않은 내용이다(역자: 과연 뭘까..되게 궁금하네..). 어쨌거나,

```
#include <unistd.h>

int gethostname(char *hostname, size_t size);
```

hostname은 문자열의 포인터이며 함수가 돌려주는 값을 담게 될 변수이다. size는 그 문자열의 크기이다.

성공적이면 0을, 에러의 경우 -1을 리턴하고 errno를 세팅한다.

---

## DNS ; whitehouse.gov - 198.137.240.100

모르는 사람을 위하여 DNS는 Domain Name Service 라는 것을 먼저 얘기 하겠다. 간결하게 얘기한다면 DNS에다가 사람이 읽을수 있는 주소를 말해주면 DNS는 bind,connect,sendto,어쨌거나 IP주소가 필요한 것들에서 사용할 수 있는 IP주소를 돌려준다. 즉 누군가가 이렇게 입력했다면

```
$ telnet whitehouse.gov
```

telnet 은 connect()에 사용하기 위해서 198.137.240.100이라는 IP주소를 찾아내게 된다. 그런데 어떻게 그렇게 하는 것인가? gethostbyname()을 사용하면 된다.

```
#include <netdb.h>
```

```
struct hostent *gethostbyname(const char *name);
```

보다시피 결과로 struct hostent의 포인터가 돌아온다. 그 구조는 아래와 같다.

```
struct hostent {
    char    *h_name;
    char    **h_aliases;
    int     h_addrtype;
    int     h_length;
    char    **h_addr_list;
};
#define h_addr h_addr_list[0]
```

각 필드에 대한 설명은 다음과 같다.

- h_name - 호스트의 공식적인 이름
- h_aliases - 호스트의 별명으로서 NULL 로 끝맺음된다.
- h_addrtype - 주소의 종류, 보통 AF_INET
- h_length - 주소의 바이트 수
- h_addr_list - 0으로 끝나는 네트워크 주소들, NBO로 되어 있다.
- h_addr - h_addr_list속의 첫번째 주소

gethostbyname()은 위의 구조체의 포인터를 돌려주게 되며 에러의 경우 NULL을 돌려준다. errno는 세팅되지 않고 h_errno가 세팅이 된다. (아래의 herror()참조)

그런데 이걸 어떻게 사용하는가? 보통 컴퓨터 매뉴얼들 처럼 독자앞에 정보를 마구 쌓아놓은 것만으로는 부족한 법이다. 이 함수는 사실 보기보다는 쓰기가 쉬운 편이다.

예제를 보자.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>

int main(int argc, char *argv[])
{
    struct hostent *h;

    if (argc != 2) {  /* error check the command line */
        fprintf(stderr,"usage: getip addressWn");
        exit(1);
    }

    if ((h=gethostbyname(argv[1])) == NULL) {  /* get the host info */
        herror("gethostbyname");
        exit(1);
    }

    printf("Host name  : %sWn", h->h_name);
    printf("IP Address : %sWn",inet_ntoa(*((struct in_addr *)h->h_addr)));

    return 0;
}
```

gethostbyname()에서는 errno가 세팅되지 않는 까닭으로 perror()를 사용할 수 없고 herror()을 사용해야 한다.

간단히 호스트의 이름을 담고 있는 스트링을 gethostbyname()함수에 넣어 줌으로써 바로 struct hostent 를 얻게 되는 것이다.

남아있는 한가지 수상한 점은 위의 방법으로 어떻게 주소를 숫자와 점으로 출력할 것인가 하는 문제이다. h->h_addr 은 문자 포인터( char *) 인데 inet_ntoa()는 변수로서 struct in_addr 을 원하기 때문이다. 따라서 h->h_addr 을 struct in_addr * 으로 형변환을 하고 결과값을 얻기 위해 다시 역참조 하면 된다는 것이다.

---

# 클라이언트-서버의 배경

요즘은 클라이언트-서버가 판치는 세상이죠~~ 네트워크에 관한 모든 것은 서버 프로세스를 요청하는 클라이언트 프로세스로서 다루어진다. 텔넷을 이용하여 23번 포트에 접속하는 (클라이언트)것은 서버프로그램(telnetd)을 작동시키게 되는 것이며 이 서버 프로그램은 들어오는 각종 신호를 받아들여서 당신의 텔넷 접속을 위하여 로그인 프롬프트를 주게 되는 것이다. 등등..

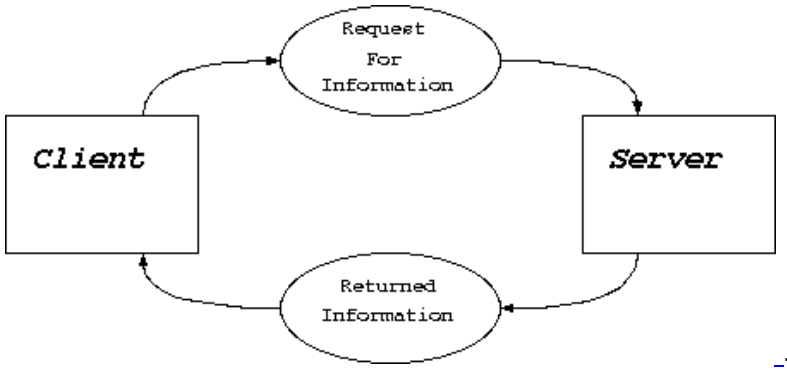


**그림2. 클라이언트-서버간의 관계**

클라이언트와 서버간의 정보 교환의 모델이 그림에 잘 나와있다.

주목할점은 클라이언트와 서버간에는 SOCK_STREAM이든, SOCK_DGRAM이든지간에 같은 것으로만 된다면 의사소통이 된다는 것이다. 좋은 예들은 telnet-telnetd, ftp-ftpd, 또는 bootp-bootpd 등이다. ftp를 쓴다면 반드시 상대편에 ftpd가 돌고 있다는 것이다.

보통 호스트에는 하나의 서버 프로그램이 돌고 있게 된다. 그리고 그 서버는 fork()를 이용하여 다중의 클라이언트를 받게 되는 것이다. 기본적인 루틴의 구조는 다음과 같다. 서버는 접속을 대기하다가 accept()를 호출하게 되며 그 때 fork()를 이용하여 자식 프로세스를 만들어내어 그 접속을 처리하게 된다. 이것이 바로 다음에 소개될 예제 서버 프로그램의 구조이다.

---

# 간단한 스트림 서버

이 서버가 하는 일은 오직 스트림 접속을 하게 되는 모든 클라이언트에게 "Hello, World!₩n"을 출력해 주는 것이다. 이 서버를 테스트하기 위해서는 하나의 윈도우에서 이 서버를 실행시켜 놓고 다른 윈도우에서 텔넷 접속을 시도해 보는 것이다.

```
$ telnet remotehostname 3490
```

hostname 은 서버 프로그램이 작동된 호스트의 이름이다.

서버 프로그램 코드

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/wait.h>

#define MYPORT 3490    /* the port users will be connecting to */

#define BACKLOG 10     /* how many pending connections queue will hold */

main()
{
    int sockfd, new_fd;  /* listen on sock_fd, new connection on new_fd */
    struct sockaddr_in my_addr;    /* my address information */
    struct sockaddr_in their_addr; /* connector's address information */
    int sin_size;
```

```
        if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
            perror("socket");
            exit(1);
        }

        my_addr.sin_family = AF_INET;         /* host byte order */
        my_addr.sin_port = htons(MYPORT);     /* short, network byte order */
        my_addr.sin_addr.s_addr = INADDR_ANY; /* auto-fill with my IP */
        bzero(&(my_addr.sin_zero), 8);        /* zero the rest of the struct */

        if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) W
                                                                    == -1) {
            perror("bind");
            exit(1);
        }

        if (listen(sockfd, BACKLOG) == -1) {
            perror("listen");
            exit(1);
        }

        while(1) {  /* main accept() loop */
            sin_size = sizeof(struct sockaddr_in);
            if ((new_fd = accept(sockfd, (struct sockaddr *)&their_addr, W
                                                    &sin_size)) == -1) {
                perror("accept");
                continue;
            }
            printf("server: got connection from %sWn", W
                                            inet_ntoa(their_addr.sin_addr));
            if (!fork()) { /* this is the child process */
                if (send(new_fd, "Hello, world!Wn", 14, 0) == -1)
                    perror("send");
                close(new_fd);
                exit(0);
            }
            close(new_fd);  /* parent doesn't need this */

            while(waitpid(-1,NULL,WNOHANG) > 0); /* clean up child processes */
        }
    }
```

이 코드는 문법상의 단순함을 위하여 하나의 커다란(내 생각에) main()에 모든 것이 들어가 있다. 만약에 이것을 잘게 잘라서 작은 여러개의 함수로 구성을 하는것이 좋다고 생각된다면 그래도 된다.

다음의 클라이언트 코드를 이용한다면 이 서버로부터 문자열을 받아 낼수도 있다.

---

# 간단한 스트림 클라이언트

이녀석은 서버보다 더 쉬운 코드이다. 이 프로그램이 하는 일은 명령행에서 지정된 주소에 3490번 포트에 접속하여 서버가 보내는 문자열을 받는 것 뿐이다.

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <errno.h>
    #include <string.h>
    #include <netdb.h>
    #include <sys/types.h>
    #include <netinet/in.h>
    #include <sys/socket.h>

    #define PORT 3490    /* the port client will be connecting to */

    #define MAXDATASIZE 100 /* max number of bytes we can get at once */

    int main(int argc, char *argv[])
    {
        int sockfd, numbytes;
        char buf[MAXDATASIZE];
```

```
        struct hostent *he;
        struct sockaddr_in their_addr; /* connector's address information */

        if (argc != 2) {
            fprintf(stderr,"usage: client hostnameWn");
            exit(1);
        }

        if ((he=gethostbyname(argv[1])) == NULL) {  /* get the host info */
            herror("gethostbyname");
            exit(1);
        }

        if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
            perror("socket");
            exit(1);
        }

        their_addr.sin_family = AF_INET;      /* host byte order */
        their_addr.sin_port = htons(PORT);    /* short, network byte order */
        their_addr.sin_addr = *((struct in_addr *)he->h_addr);
        bzero(&(their_addr.sin_zero), 8);     /* zero the rest of the struct */

        if (connect(sockfd, (struct sockaddr *)&their_addr, W
                                              sizeof(struct sockaddr)) == -1) {
            perror("connect");
            exit(1);
        }

        if ((numbytes=recv(sockfd, buf, MAXDATASIZE, 0)) == -1) {
            perror("recv");
            exit(1);
        }

        buf[numbytes] = 'W0';

        printf("Received: %s",buf);

        close(sockfd);

        return 0;
    }
```

이 클라이언트를 작동하기에 앞서서 서버를 작동시켜놓지 않았다면 connect()함수는 "Connection refused"를 돌려주게 될것이다. 쓸만하군!

---

# 데이터그램 소켓

이에 관해서는 그다지 얘기할 것이 많지 않다. 따라서 그냥 두개의 프로그램을 보여 주겠다.

listener는 호스트에 앉아서 4950포트에 들어오는 데이터 패킷을 기다린다. talker는 지정된 호스트의 그 포트로 뭐든지 간에 사용자가 입력한 데이터를 보낸다.

listener.c

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <errno.h>
    #include <string.h>
    #include <sys/types.h>
    #include <netinet/in.h>
    #include <sys/socket.h>
    #include <sys/wait.h>

    #define MYPORT 4950    /* the port users will be connecting to */

    #define MAXBUFLEN 100

    main()
    {
```

```
        int sockfd;
        struct sockaddr_in my_addr;    /* my address information */
        struct sockaddr_in their_addr; /* connector's address information */
        int addr_len, numbytes;
        char buf[MAXBUFLEN];

        if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
            perror("socket");
            exit(1);
        }

        my_addr.sin_family = AF_INET;         /* host byte order */
        my_addr.sin_port = htons(MYPORT);     /* short, network byte order */
        my_addr.sin_addr.s_addr = INADDR_ANY; /* auto-fill with my IP */
        bzero(&(my_addr.sin_zero), 8);        /* zero the rest of the struct */

        if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) W
                                                                    == -1) {
            perror("bind");
            exit(1);
        }

        addr_len = sizeof(struct sockaddr);
        if ((numbytes=recvfrom(sockfd, buf, MAXBUFLEN, 0, W
                          (struct sockaddr *)&their_addr, &addr_len)) == -1) {
            perror("recvfrom");
            exit(1);
        }

        printf("got packet from %sWn",inet_ntoa(their_addr.sin_addr));
        printf("packet is %d bytes longWn",numbytes);
        buf[numbytes] = 'W0';
        printf("packet contains W"%sW"Wn",buf);

        close(sockfd);
    }
```

결국 socket()를 호출할 때 SOCK_DGRAM을 사용하게 된것을 주의하고, listen()이나 accept()를 사용하지 않은것도 주의해 봐야 한다. 이 코드가 바로 비연결 데이터그램 소켓의 자랑스러운 사용예인 것이다.

talker.c

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <errno.h>
    #include <string.h>
    #include <sys/types.h>
    #include <netinet/in.h>
    #include <netdb.h>
    #include <sys/socket.h>
    #include <sys/wait.h>

    #define MYPORT 4950    /* the port users will be connecting to */

    int main(int argc, char *argv[])
    {
        int sockfd;
        struct sockaddr_in their_addr; /* connector's address information */
        struct hostent *he;
        int numbytes;

        if (argc != 3) {
            fprintf(stderr,"usage: talker hostname messageWn");
            exit(1);
        }

        if ((he=gethostbyname(argv[1])) == NULL) {  /* get the host info */
            herror("gethostbyname");
            exit(1);
        }

        if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
            perror("socket");
            exit(1);
```

```
        }

        their_addr.sin_family = AF_INET;      /* host byte order */
        their_addr.sin_port = htons(MYPORT);  /* short, network byte order */
        their_addr.sin_addr = *((struct in_addr *)he->h_addr);
        bzero(&(their_addr.sin_zero), 8);     /* zero the rest of the struct */

        if ((numbytes=sendto(sockfd, argv[2], strlen(argv[2]), 0, W
             (struct sockaddr *)&their_addr, sizeof(struct sockaddr))) == -1) {
            perror("sendto");
            exit(1);
        }

        printf("sent %d bytes to %sWn",numbytes,inet_ntoa(their_addr.sin_addr));

        close(sockfd);

        return 0;
    }
```

이것이 다다. listener를 한 호스트에서 실행 시키고 다른 곳에서 talker를 실행시킨다. 핵가족시대에 어울리는 가족용 오락이 될수도...

앞에서도 얘기했었지만 한가지 작은 내용을 더 말해야 할것 같다. 만약 talker에서 connect()를 호출해서 연결을 했다면 그 다음부터는 sendto(), recvfrom()이 아니라 그냥 send().recv()를 사용해도 된다는 것이다. 전달되어야 하는 호스트의 주소는 connect()에 지정된 주소가 사용되게 된다.

---

# 블로킹

블로킹. 아마 들어봤겠지. 그런데 도대체 그게 뭘까? 사실 "잠들다"의 기술용어에 불과한 것이다. 아마도 listener를 실행시키면서 눈치를 챘겠지만 그 프로그램은 그저 앉아서 데이터 패킷이 올때까지 기다리는 것이다. 잠자면서.. recvfrom()을 호출했는데 데이터가 들어온 것이 없다면? 바로 뭔가 데이터가 들어올 때까지 블로킹이 되는 것이다(그냥 거기서 자고 있는 것이다.).

많은 함수들이 블로킹이 된다. accept()는 블록이 된다. recv*()종류들이 모두 블록이 된다. 그들이 이렇게 할 수 있는 이유는 그렇게 할 수 있도록 허락을 받았기 때문이다. 처음에 socket()으로 소켓이 만들어질때 커널이 블록 가능하도록 세팅을 했기 때문이다. 만일 블록할수 없도록 세팅하려면 fcntl()을 사용한다.

```
#include <unistd.h>
#include <fcntl.h>
.
.
sockfd = socket(AF_INET, SOCK_STREAM, 0);
fcntl(sockfd, F_SETFL, O_NONBLOCK);
.
.
```

소켓을 블록할수 없도록 세팅함으로써 정보를 추출하는 데에 효과적으로 socket을 이용할 수 있다. 만일 데이터가 접수되지 않은 소켓에서 데이터를 읽으려고 시도한다면 -1을 결과로 돌려주고 errno를 EWOULDBLOCK 으로 세팅하게 된다.

일반적으로는 이런 식으로 정보를 뽑아 내는 것은 별로 좋은 방식은 아니다. 만일 들어오는 데이터를 감시하기 위하여 이런 방식으로 바쁘게 데이터를 찾는 루틴을 만든다면 이는 CPU 시간을 소모하게 되는 것이다. 구식이다. 보다 멋진 방법은 다음절에 나오는 select()를 사용하여 데이터를 기다리는 식이다.

---

# select() ; 동기화된 중복 입출력. 대단하군!

이건 뭔가 좀 이상한 함수이다. 그러나 상당히 유용하므로 잘 읽어보기 바란다. 다음 상황을 가정해 보자. 지금 서버를 돌리고 있으며 이미 연결된 소켓에서 데이터가 들어오는 것을 기다리고 있다고 하자.

문제없지, 그냥 accept()하고 recv()몇개면 될텐데.. 서둘지 말지어다, 친구. 만일 accept()에서 블로킹이 된다면? 동시에 어떻게 recv()를 쓸 것인가? 블로킹 못하게 세팅한다고? CPU시간을 낭비하지 말라니까. 그러면 어떻게?

더이상 떠들지 말고 다음을 보여주겠다.

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select(int numfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

이 함수는 화일 기술자의 "집합", 특별히 readfds,writefds,exceptfds등을 관리한다. 만일 일반적인 입력이나 소켓 기술자로부터 읽어 들일수 있는가를 확인하려면 단지 화일 기술자 0과 sockfd를 readfds에 더해주기만 하면 된다. numfds는 가장 높은 파일 기술자에다가 1을 더해서 지정해야 하며 이번 예제에서는 정규 입력의 0보다 확실히 크게 하기 위해서 sockfd+1 을 지정해야 한다.

select()의 결과값이 나올때 readfs는 선택한 파일 기술자 중에 어떤 것이 읽기 가능한가를 반영할 수 있도록 수정되며 FD_ISSET() 매크로를 이용하여 체크할 수 있다.

너무 멀리 나가기 전에 이 "집합"들을 어떻게 관리하는 가에 대해서 얘기를 해야 할것 같다. 각각의 "집합"은 fd_set형이며 다음의 매크로들로 이를 제어할 수 있다.

- `FD_ZERO(fd_set *set)` - 파일기술자 집합을 소거한다.
- `FD_SET(int fd, fd_set *set)` - fd 를 set에 더해준다.
- `FD_CLR(int fd, fd_set *set)` - fd 를 set에서 빼준다.
- `FD_ISSET(int fd, fd_set *set)` - fd가 set안에 있는지 확인한다.

끝으로 이 수상한 struct timeval은 또 무엇인가? 아마도 누군가가 어떤 데이터를 보내는 것을 무한정 기다리기를 원치는 않을 것이다. 특정시간마다 아무일도 안 벌어지더라도 "현재 진행중..."이라는 메시지를 터미널에 출력시키기라도 원할 것이다. 이 구조체는 그 시간간격을 정의하기 위해서 사용되는 것이다. 이 시간이 초과되고 그 때까지 select()가 아무런 변화를 감지하지 못한 경우라면 결과를 돌려주고 다음 작업을 진행 할수 있도록 해준다.

struct timeval의 구조는 다음과 같다.

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

기다릴 시간의 초를 지정하려면 그냥 tv_sec에 지정하면 된다. tv_usec에는 마이크로 초를 지정한다. 밀리초가 아니고 마이크로초이다. 마이크로초는 백만분의 일초이다. 그런데 왜 usec인가? u는 그리스 문자의 Mu를 닮았고 이는 마이크로를 의미하는데 사용된다. 함수가 끝날때 timeout에 남은 시간이 기록될수도 있으며 이 내용은 유닉스마다 다르기는 하다.

와우~ 마이크로 초 단위의 타이머를 가지게 되었군! 만일 timeval에 필드들을 0으로 채우면 select()는 즉시 결과를 돌려주며 현재 set들의 내용을 즉시 알려주게 된다. timeout을 NULL로 세팅하면 결코 끝나지 않고 계속 파일 기술자가 준비되는 것을 기다리게 되며 끝으로 특정한 set에 변화에 관심이 없다면 그 항목을 NULL로 지정하면 된다.

다음은 정규 입력에 무언가 나타날때까지 2.5초를 기다리는 코드이다.

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

#define STDIN 0  /* file descriptor for standard input */

main()
{
    struct timeval tv;
    fd_set readfds;

    tv.tv_sec = 2;
    tv.tv_usec = 500000;
```

```
        FD_ZERO(&readfds);
        FD_SET(STDIN, &readfds);

        /* don't care about writefds and exceptfds: */
        select(STDIN+1, &readfds, NULL, NULL, &tv);

        if (FD_ISSET(STDIN, &readfds))
            printf("A key was pressed!₩n");
        else
            printf("Timed out.₩n");
    }
```

만일 한줄씩 버퍼링하는 터미널이라면 엔터키를 치지 않는 이상은 그냥 타임아웃에 걸릴것이다.

이제 아마도 이 훌륭한 방법을 데이터그램 소켓에서 데이터를 기다리는 데에 사용할수 있으리라고 생각할 것이다. 맞다. 그럴 수도 있다. 어떤 유닉스에서는 이 방법이 되지만 안되는 것도 있다. 하고자 하는 내용에 대해서는 아마도 맨페이지를 참조해야 할 것이다.

select()에 관한 마지막 얘기는 listen()이 된 소켓이 있다면 이 방법을 이용하여 소켓 기술자를 readfds에 첨가하는 방식으로 새로운 연결이 있었는가를 확인할 수도 있다는 것이다.

이것이 select()에 대한 짧은 검토였다.

# 참고사항

여기까지 와서는 아마 좀더 새로운 다른 것은 없는가 할것이다. 또 어디서 다른 무언가를 더 찾을수 있는가를 알고자 할것이다.

초보자라면 다음의 맨페이지를 참고하는 것도 좋다.

- socket()
- bind()
- connect()
- listen()
- accept()
- send()
- recv()
- sendto()
- recvfrom()
- close()
- shutdown()
- getpeername()
- getsockname()
- gethostbyname()
- gethostbyaddr()
- getprotobyname()
- fcntl()
- select()
- perror()

다음 책들도 도움이 될것이다. books:

**Internetworking with TCP/IP, volumes I-III**
by Douglas E. Comer and David L. Stevens.
Published by Prentice Hall.
Second edition ISBNs: 0-13-468505-9, 0-13-472242-6, 0-13-474222-2.
There is a third edition of this set which covers IPv6 and IP over ATM.

**Using C on the UNIX System**
by David A. Curry.
Published by O'Reilly & Associates, Inc.
ISBN 0-937175-23-4.

**TCP/IP Network Administration**
by Craig Hunt.
Published by O'Reilly & Associates, Inc.
ISBN 0-937175-82-X.

**TCP/IP Illustrated, volumes 1-3**
by W. Richard Stevens and Gary R. Wright.
Published by Addison Wesley.
ISBNs: 0-201-63346-9, 0-201-63354-X, 0-201-63495-3.

**Unix Network Programming**
by W. Richard Stevens.
Published by Prentice Hall.
ISBN 0-13-949876-1.

웹상에는 다음과 같은 것들이 있을것이다.

**BSD Sockets: A Quick And Dirty Primer** (http://www.cs.umn.edu/~bentlema/unix/--has other great Unix system programming info, too!)

**Client-Server Computing** (http://pandonia.canberra.edu.au/ClientServer/socket.html)

**Intro to TCP/IP** (gopher) (gopher://gopher-chem.ucdavis.edu/11/Index/Internet_aw/Intro_the_Internet/intro.to.ip/)

**Internet Protocol Frequently Asked Questions** (France) (http://web.cnam.fr/Network/TCP-IP/)

**The Unix Socket FAQ** (http://www.ibrado.com/sock-faq/)

끔찍하지만..RFC도 봐야 하겠다.

**RFC-768** -- The User Datagram Protocol (UDP) (ftp://nic.ddn.mil/rfc/rfc768.txt)

**RFC-791** -- The Internet Protocol (IP) (ftp://nic.ddn.mil/rfc/rfc791.txt)

**RFC-793** -- The Transmission Control Protocol (TCP) (ftp://nic.ddn.mil/rfc/rfc793.txt)

**RFC-854** -- The Telnet Protocol (ftp://nic.ddn.mil/rfc/rfc854.txt)

**RFC-951** -- The Bootstrap Protocol (BOOTP) (ftp://nic.ddn.mil/rfc/rfc951.txt)

**RFC-1350** -- The Trivial File Transfer Protocol (TFTP) (ftp://nic.ddn.mil/rfc/rfc1350.txt)

---

# 주의사항 및 연락처

이상이 전부이며 문서상에서 크게 틀린 곳이 없기만을 바랄 뿐이다. 하지만 실수는 항상 있는 법이다.

만약 실수가 있다면 부정확한 정보를 주어 헷갈리게 만듯것에 대하여 사과하지만 사실상 나한테 책임을 물을수는 없다. 이 얘기는 법적인 경고이며 사실상 이 모든 글들이 몽땅 거짓말 일수도 있는 것이다.

그래도 설마 그렇지는 않을 것이다. 사실 난 이 모든것들 때문에 상당히 많은 시간을 소모했고 윈도우용 TCP/IP네트워크 유틸리티(예를 들어 텔넷등)을 방학숙제로 했었다. 난 소켓의 신이 아니라 그냥 보통 사람일 뿐이다.

그건 그렇고 생산적인 (혹은 파괴적이라도) 비평이 있는 분은 beej@ecst.csuchico.edu 앞으로 메일을 주기 바란다. 참고하여 고쳐나가도록 노력을 해 보겠다.

왜 이 일을 했는가 궁금하다면, 돈벌려고 했다. 하하~ 사실은 아니고 많은 사람들이 소켓에 관련된 질문을 해 대는 바람에 그들에게 이 내용을 웹에 올리려고 생각중이라고 말했더니 "바로 그거야~"라고들 해서 썼다. 아무리 고생해서 얻은 정보라도 만일 다른 사람과 공유하지 않는다면 쓰레기일 뿐이라고 생각한다. WWW는 바로 적당한 수단이 된 것 뿐이다. 다른 사람도 이런 정보의 제공이 가능하다면 이렇게 해주길 바란다.

끝났다. 프로그램이나 짜러가자. ;-)

번역한 사람의 말: 우연히 이 글을 발견하게 되어 번역을 하고 보니 나름대로 가치가 있어 보여서 홈페이지에 올려 놓았습니다. 번역상의 실수가 있었다면 사과드리며 지적해 주신다면 고쳐 나가겠습니다. 좋은 프로그램을 만드는데에 이 글이 작으나마 도움이 되길 바랍니다.

---